MetalPerformanceShaders.framework

MetalPerformanceShaders-59

Generated by Doxygen 1.8.10

Tue Jul 12 2016 13:07:21

Contents

1 Metal Shaders - High Performance Kernels on Metal **1**

1.1 Introduction . . . 1

1.2 Data containers . . . 1

1.2.1 MTLTextures and MTLBuffers . . . 1

1.2.2 MPSImages . . . 1

1.2.3 MPSTemporaryImages . . . 2

1.3 The MPSKernel . . . 2

1.3.1 MPS{Unary/Binary}ImageKernel properties . . . 3

1.3.1.1 MPSKernel clipRect . . . 3

1.3.1.2 MPSOffset . . . 3

1.3.1.3 MPSKernelEdgeMode . . . 3

1.3.1.4 MPSKernelOptions . . . 4

1.4 Available MPSKernels . . . 4

1.4.1 Image Convolution . . . 4

1.4.1.1 The Image Convolution Kernel . . . 4

1.4.1.2 The Box, Tent and Gaussian Filters . . . 5

1.4.1.3 Laplacian and Unsharp Mask Filters . . . 5

1.4.1.4 Sobel Edge detection . . . 5

1.4.1.5 Other Filters . . . 5

1.4.1.6 Separable Convolution . . . 6

1.4.1.7 Convolutions in MPS . . . 6

1.4.2 Morphology . . . 6

1.4.3 Histogram . . . 6

1.4.4 Image Median . . . 7

1.4.5 Image Resampling . . . 7

1.4.6 Image Threshold . . . 7

1.4.7 Convolutional Neural Networks . . . 8

1.5 MPS API validation . . . 9

1.6 How to Add MetalPerformanceShaders.framework to your project . . . 9

1.7 How to Determine if MPS Works on Your Device . . . 9

1.8 In Place Operation . . . 9

1.8.1 MPSCopyAllocator . . . 10
1.9 Sample Code . . . 10
1.10 MPS Tuning Hints . . . 11

2 Hierarchical Index **13**
2.1 Class Hierarchy . . . 13

3 Class Index **15**
3.1 Class List . . . 15

4 Class Documentation **17**
4.1 MPSBinaryImageKernel Class Reference . . . 17
4.1.1 Detailed Description . . . 17
4.1.2 Method Documentation . . . 18
4.1.2.1 encodeToCommandBuffer:inPlacePrimaryTexture:secondaryTexture:fallback↩ CopyAllocator:(nonnull id< MTLCommandBuffer > commandBuffer,[inPlace↩ PrimaryTexture] __nonnull id< MTLTexture > __strong *__nonnull inPlace↩ PrimaryTexture,[secondaryTexture] nonnull id< MTLTexture > secondary↩ Texture,[fallbackCopyAllocator] nullable MPSCopyAllocator copyAllocator) . . . 18
4.1.2.2 encodeToCommandBuffer:primaryTexture:inPlaceSecondaryTexture:fallback↩ CopyAllocator:(nonnull id< MTLCommandBuffer > commandBuffer,[primary↩ Texture] nonnull id< MTLTexture > primaryTexture,[inPlaceSecondary↩ Texture] __nonnull id< MTLTexture > __strong *__nonnull inPlaceSecondary↩ Texture,[fallbackCopyAllocator] nullable MPSCopyAllocator copyAllocator) . . . 18
4.1.2.3 encodeToCommandBuffer:primaryTexture:secondaryTexture:destination↩ Texture:(nonnull id< MTLCommandBuffer > commandBuffer,[primaryTexture] nonnull id< MTLTexture > primaryTexture,[secondaryTexture] nonnull id< M↩ TLTexture > secondaryTexture,[destinationTexture] nonnull id< MTLTexture > destinationTexture) . . . 19
4.1.2.4 primarySourceRegionForDestinationSize:(MTLSize destinationSize) . . . 19
4.1.2.5 secondarySourceRegionForDestinationSize:(MTLSize destinationSize) . . . 20
4.1.3 Property Documentation . . . 20
4.1.3.1 clipRect . . . 20
4.1.3.2 primaryEdgeMode . . . 21
4.1.3.3 primaryOffset . . . 21
4.1.3.4 secondaryEdgeMode . . . 21
4.1.3.5 secondaryOffset . . . 21
4.2 MPSCNNConvolution Class Reference . . . 21
4.2.1 Detailed Description . . . 22
4.2.2 Method Documentation . . . 22
4.2.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 22
4.2.2.2 initWithDevice:convolutionDescriptor:kernelWeights:biasTerms:flags:(nonnull id< MTLDevice > device,[convolutionDescriptor] const MPSCNNConvolution↩ Descriptor *__nonnull convolutionDescriptor,[kernelWeights] const float *_↩ _nonnull kernelWeights,[biasTerms] const float *__nullable biasTerms,[flags] MPSCNNConvolutionFlags NS_DESIGNATED_INITIALIZER) . . . 23

4.2.3 Property Documentation . . . 24
4.2.3.1 groups . . . 24
4.2.3.2 inputFeatureChannels . . . 24
4.2.3.3 kernelHeight . . . 24
4.2.3.4 kernelWidth . . . 24
4.2.3.5 neuron . . . 24
4.2.3.6 outputFeatureChannels . . . 24
4.2.3.7 strideInPixelsX . . . 24
4.2.3.8 strideInPixelsY . . . 25
4.3 MPSCNNConvolutionDescriptor Class Reference . . . 25
4.3.1 Detailed Description . . . 25
4.3.2 Method Documentation . . . 25
4.3.2.1 cnnConvolutionDescriptorWithKernelWidth:kernelHeight:inputFeatureChannels↩ :outputFeatureChannels:neuronFilter:(NSUInteger kernelWidth,[kernelHeight] NSUInteger kernelHeight,[inputFeatureChannels] NSUInteger inputFeature↩ Channels,[outputFeatureChannels] NSUInteger outputFeatureChannels,[neuron↩ Filter] const MPSCNNNeuron ∗__nullable neuronFilter) . . . 25
4.3.3 Property Documentation . . . 26
4.3.3.1 groups . . . 26
4.3.3.2 inputFeatureChannels . . . 26
4.3.3.3 kernelHeight . . . 26
4.3.3.4 kernelWidth . . . 26
4.3.3.5 neuron . . . 26
4.3.3.6 outputFeatureChannels . . . 26
4.3.3.7 strideInPixelsX . . . 26
4.3.3.8 strideInPixelsY . . . 27
4.4 MPSCNNCrossChannelNormalization Class Reference . . . 27
4.4.1 Detailed Description . . . 27
4.4.2 Method Documentation . . . 28
4.4.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 28
4.4.2.2 initWithDevice:kernelSize:(nonnull id< MTLDevice > device,[kernelSize] NSU↩ Integer NS_DESIGNATED_INITIALIZER) . . . 29
4.4.3 Property Documentation . . . 29
4.4.3.1 alpha . . . 29
4.4.3.2 beta . . . 29
4.4.3.3 delta . . . 29
4.4.3.4 kernelSize . . . 29
4.5 MPSCNNFullyConnected Class Reference . . . 29
4.5.1 Detailed Description . . . 30
4.5.2 Method Documentation . . . 30
4.5.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 30

4.5.2.2 initWithDevice:convolutionDescriptor:kernelWeights:biasTerms:flags:(nonnull id< MTLDevice > device,[convolutionDescriptor] const MPSCNNConvolution↩ Descriptor *__nonnull fullyConnectedDescriptor,[kernelWeights] const float *↩ __nonnull kernelWeights,[biasTerms] const float *__nullable biasTerms,[flags] MPSCNNConvolutionFlags NS_DESIGNATED_INITIALIZER) 31

4.6 MPSCNNKernel Class Reference 32

4.6.1 Detailed Description 33

4.6.2 Method Documentation 33

4.6.2.1 encodeToCommandBuffer:sourceImage:destinationImage:(nonnull id< M↩ TLCommandBuffer > commandBuffer,[sourceImage] MPSImage *__↩ nonnull sourceImage,[destinationImage] encode(commandBuffer:sourceImage↩ :destinationImage:) MPS_SWIFT_NAME) 33

4.6.2.2 sourceRegionForDestinationSize:(sourceRegion(destinationSize:) MPS_SWIF↩ T_NAME) 34

4.6.3 Property Documentation 34

4.6.3.1 clipRect 34

4.6.3.2 destinationFeatureChannelOffset 34

4.6.3.3 edgeMode 35

4.6.3.4 offset 35

4.7 MPSCNNLocalContrastNormalization Class Reference 35

4.7.1 Detailed Description 36

4.7.2 Method Documentation 36

4.7.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) 36

4.7.2.2 initWithDevice:kernelWidth:kernelHeight:(nonnull id< MTLDevice > device,[kernel↩ Width] NSUInteger kernelWidth,[kernelHeight] NSUInteger NS_DESIGNATED↩ _INITIALIZER) 36

4.7.3 Property Documentation 37

4.7.3.1 alpha 37

4.7.3.2 beta 37

4.7.3.3 delta 37

4.7.3.4 kernelHeight 37

4.7.3.5 kernelWidth 37

4.7.3.6 p0 37

4.7.3.7 pm 37

4.7.3.8 ps 37

4.8 MPSCNNLogSoftMax Class Reference 37

4.8.1 Detailed Description 38

4.9 MPSCNNNeuron Class Reference 38

4.9.1 Detailed Description 38

4.10 MPSCNNNeuronAbsolute Class Reference 39

4.10.1 Detailed Description 39

4.10.2 Method Documentation 39

4.10.2.1 initWithDevice:(nonnull id< MTLDevice > NS_DESIGNATED_INITIALIZER) . . 39

4.11 MPSCNNNeuronLinear Class Reference . . . 39
4.11.1 Detailed Description . . . 40
4.11.2 Method Documentation . . . 40
4.11.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 40
4.11.2.2 initWithDevice:a:b:(nonnull id< MTLDevice > device,[a] float a,[b] float NS_DE↩ SIGNATED_INITIALIZER) . . . 40
4.12 MPSCNNNeuronReLU Class Reference . . . 41
4.12.1 Detailed Description . . . 41
4.12.2 Method Documentation . . . 42
4.12.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 42
4.12.2.2 initWithDevice:a:(nonnull id< MTLDevice > device,[a] float NS_DESIGNATED↩ _INITIALIZER) . . . 43
4.13 MPSCNNNeuronSigmoid Class Reference . . . 43
4.13.1 Detailed Description . . . 44
4.13.2 Method Documentation . . . 44
4.13.2.1 initWithDevice:(nonnull id< MTLDevice > NS_DESIGNATED_INITIALIZER) . . . 44
4.14 MPSCNNNeuronTanH Class Reference . . . 44
4.14.1 Detailed Description . . . 45
4.14.2 Method Documentation . . . 45
4.14.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 45
4.14.2.2 initWithDevice:a:b:(nonnull id< MTLDevice > device,[a] float a,[b] float NS_DE↩ SIGNATED_INITIALIZER) . . . 45
4.15 MPSCNNPooling Class Reference . . . 45
4.15.1 Detailed Description . . . 46
4.15.2 Method Documentation . . . 46
4.15.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 46
4.15.2.2 initWithDevice:kernelWidth:kernelHeight:(nonnull id< MTLDevice > device,[kernel↩ Width] NSUInteger kernelWidth,[kernelHeight] NSUInteger kernelHeight) . . . 47
4.15.2.3 initWithDevice:kernelWidth:kernelHeight:strideInPixelsX:strideInPixelsY:(nonnull id< MTLDevice > device,[kernelWidth] NSUInteger kernelWidth,[kernelHeight] NSUInteger kernelHeight,[strideInPixelsX] NSUInteger strideInPixelsX,[stride↩ InPixelsY] NSUInteger NS_DESIGNATED_INITIALIZER) . . . 48
4.15.3 Property Documentation . . . 48
4.15.3.1 kernelHeight . . . 48
4.15.3.2 kernelWidth . . . 48
4.15.3.3 strideInPixelsX . . . 48
4.15.3.4 strideInPixelsY . . . 48
4.16 MPSCNNPoolingAverage Class Reference . . . 48
4.16.1 Detailed Description . . . 49
4.17 MPSCNNPoolingMax Class Reference . . . 49
4.17.1 Detailed Description . . . 50
4.18 MPSCNNSoftMax Class Reference . . . 50

4.18.1 Detailed Description . . . 50
4.19 MPSCNNSpatialNormalization Class Reference . . . 50
4.19.1 Detailed Description . . . 51
4.19.2 Method Documentation . . . 51
4.19.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 51
4.19.2.2 initWithDevice:kernelWidth:kernelHeight:(nonnull id< MTLDevice > device,[kernel↩ Width] NSUInteger kernelWidth,[kernelHeight] NSUInteger NS_DESIGNATED↩ _INITIALIZER) . . . 52
4.19.3 Property Documentation . . . 53
4.19.3.1 alpha . . . 53
4.19.3.2 beta . . . 53
4.19.3.3 delta . . . 53
4.19.3.4 kernelHeight . . . 53
4.19.3.5 kernelWidth . . . 53
4.20 MPSImage Class Reference . . . 53
4.20.1 Detailed Description . . . 54
4.20.2 Method Documentation . . . 55
4.20.2.1 initWithDevice:imageDescriptor:(nonnull id< MTLDevice > device,[image↩ Descriptor] const MPSImageDescriptor *__nonnull imageDescriptor) . . . 55
4.20.2.2 initWithTexture:featureChannels:(nonnull id< MTLTexture > texture,[feature↩ Channels] NSUInteger featureChannels) . . . 55
4.20.2.3 setPurgeableState:(MPSPurgeableState state) . . . 56
4.20.3 Property Documentation . . . 56
4.20.3.1 device . . . 56
4.20.3.2 featureChannels . . . 56
4.20.3.3 height . . . 56
4.20.3.4 label . . . 56
4.20.3.5 numberOfImages . . . 56
4.20.3.6 pixelFormat . . . 56
4.20.3.7 pixelSize . . . 57
4.20.3.8 precision . . . 57
4.20.3.9 texture . . . 57
4.20.3.10 textureType . . . 57
4.20.3.11 usage . . . 57
4.20.3.12 width . . . 57
4.21 MPSImageAreaMax Class Reference . . . 57
4.21.1 Detailed Description . . . 58
4.21.2 Method Documentation . . . 58
4.21.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 58
4.21.2.2 initWithDevice:kernelWidth:kernelHeight:(nonnull id< MTLDevice > device,[kernel↩ Width] NSUInteger kernelWidth,[kernelHeight] NSUInteger NS_DESIGNATED↩ _INITIALIZER) . . . 59

4.21.3 Property Documentation . . . 60
4.21.3.1 kernelHeight . . . 60
4.21.3.2 kernelWidth . . . 60
4.22 MPSImageAreaMin Class Reference . . . 60
4.22.1 Detailed Description . . . 60
4.23 MPSImageBox Class Reference . . . 61
4.23.1 Detailed Description . . . 61
4.23.2 Method Documentation . . . 61
4.23.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 61
4.23.2.2 initWithDevice:kernelWidth:kernelHeight:(nonnull id< MTLDevice > device,[kernel↩ Width] NSUInteger kernelWidth,[kernelHeight] NSUInteger NS_DESIGNATED↩ _INITIALIZER) . . . 62
4.23.3 Property Documentation . . . 62
4.23.3.1 kernelHeight . . . 62
4.23.3.2 kernelWidth . . . 62
4.24 MPSImageConversion Class Reference . . . 62
4.24.1 Detailed Description . . . 63
4.24.2 Method Documentation . . . 63
4.24.2.1 initWithDevice:srcAlpha:destAlpha:backgroundColor:conversionInfo:(nonnull id< MTLDevice > device,[srcAlpha] MPSAlphaType srcAlpha,[destAlpha] MP↩ SAlphaType destAlpha,[backgroundColor] nullable CGFloat *background↩ Color,[conversionInfo] nullable CGColorConversionInfoRef conversionInfo) . . . 63
4.24.3 Property Documentation . . . 63
4.24.3.1 destinationAlpha . . . 63
4.24.3.2 sourceAlpha . . . 63
4.25 MPSImageConvolution Class Reference . . . 64
4.25.1 Detailed Description . . . 64
4.25.2 Method Documentation . . . 65
4.25.2.1 initWithDevice:kernelWidth:kernelHeight:weights:(nonnull id< MTLDevice > device,[kernelWidth] NSUInteger kernelWidth,[kernelHeight] NSUInteger kernelHeight,[weights] const float *__nonnull NS_DESIGNATED_INITIALIZ↩ ER) . . . 65
4.25.3 Property Documentation . . . 65
4.25.3.1 bias . . . 65
4.25.3.2 kernelHeight . . . 65
4.25.3.3 kernelWidth . . . 65
4.26 MPSImageDescriptor Class Reference . . . 65
4.26.1 Detailed Description . . . 66
4.26.2 Method Documentation . . . 66
4.26.2.1 imageDescriptorWithChannelFormat:width:height:featureChannels:(MPS↩ ImageFeatureChannelFormat channelFormat,[width] NSUInteger width,[height] NSUInteger height,[featureChannels] NSUInteger featureChannels) . . . 66

4.26.2.2 imageDescriptorWithChannelFormat:width:height:featureChannels:numberOf↩ Images:usage:(MPSImageFeatureChannelFormat channelFormat,[width] N↩ SUInteger width,[height] NSUInteger height,[featureChannels] NSUInteger featureChannels,[numberOfImages] NSUInteger numberOfImages,[usage] MT↩ LTextureUsage usage) . . . 66
4.26.3 Property Documentation . . . 67
4.26.3.1 channelFormat . . . 67
4.26.3.2 cpuCacheMode . . . 67
4.26.3.3 featureChannels . . . 67
4.26.3.4 height . . . 67
4.26.3.5 numberOfImages . . . 67
4.26.3.6 pixelFormat . . . 67
4.26.3.7 storageMode . . . 67
4.26.3.8 usage . . . 67
4.26.3.9 width . . . 67
4.27 MPSImageDilate Class Reference . . . 67
4.27.1 Detailed Description . . . 68
4.27.2 Method Documentation . . . 68
4.27.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 68
4.27.2.2 initWithDevice:kernelWidth:kernelHeight:values:(nonnull id< MTLDevice > device,[kernelWidth] NSUInteger kernelWidth,[kernelHeight] NSUInteger kernelHeight,[values] const float *__nonnull NS_DESIGNATED_INITIALIZER) . 69
4.27.3 Property Documentation . . . 69
4.27.3.1 kernelHeight . . . 69
4.27.3.2 kernelWidth . . . 69
4.28 MPSImageErode Class Reference . . . 69
4.28.1 Detailed Description . . . 70
4.29 MPSImageGaussianBlur Class Reference . . . 70
4.29.1 Detailed Description . . . 71
4.29.2 Method Documentation . . . 71
4.29.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 71
4.29.2.2 initWithDevice:sigma:(nonnull id< MTLDevice > device,[sigma] float NS_DESI↩ GNATED_INITIALIZER) . . . 71
4.29.3 Property Documentation . . . 72
4.29.3.1 sigma . . . 72
4.30 MPSImageGaussianPyramid Class Reference . . . 72
4.30.1 Detailed Description . . . 72
4.31 MPSImageHistogram Class Reference . . . 72
4.31.1 Detailed Description . . . 73
4.31.2 Method Documentation . . . 73

4.31.2.1 encodeToCommandBuffer:sourceTexture:histogram:histogramOffset:(nonnull id< MTLCommandBuffer > commandBuffer,[sourceTexture] nonnull id< MTL↩ Texture > source,[histogram] nonnull id< MTLBuffer > histogram,[histogram↩ Offset] NSUInteger histogramOffset) . . . 73

4.31.2.2 histogramSizeForSourceFormat:(MTLPixelFormat sourceFormat) . . . 73

4.31.2.3 initWithDevice:histogramInfo:(nonnull id< MTLDevice > device,[histogramInfo] const MPSImageHistogramInfo *__nonnull NS_DESIGNATED_INITIALIZER) . . 74

4.31.3 Property Documentation . . . 74

4.31.3.1 clipRectSource . . . 74

4.31.3.2 histogramInfo . . . 74

4.31.3.3 zeroHistogram . . . 74

4.32 MPSImageHistogramEqualization Class Reference . . . 74

4.32.1 Detailed Description . . . 75

4.32.2 Method Documentation . . . 75

4.32.2.1 encodeTransformToCommandBuffer:sourceTexture:histogram:histogram↩ Offset:(nonnull id< MTLCommandBuffer > commandBuffer,[sourceTexture] nonnull id< MTLTexture > source,[histogram] nonnull id< MTLBuffer > histogram,[histogramOffset] NSUInteger histogramOffset) . . . 75

4.32.2.2 initWithDevice:histogramInfo:(nonnull id< MTLDevice > device,[histogramInfo] const MPSImageHistogramInfo *__nonnull NS_DESIGNATED_INITIALIZER) . . 76

4.32.3 Property Documentation . . . 76

4.32.3.1 histogramInfo . . . 76

4.33 MPSImageHistogramInfo Struct Reference . . . 76

4.33.1 Detailed Description . . . 77

4.33.2 Member Data Documentation . . . 77

4.33.2.1 histogramForAlpha . . . 77

4.33.2.2 maxPixelValue . . . 77

4.33.2.3 minPixelValue . . . 77

4.33.2.4 numberOfHistogramEntries . . . 77

4.34 MPSImageHistogramSpecification Class Reference . . . 77

4.34.1 Detailed Description . . . 78

4.34.2 Method Documentation . . . 78

4.34.2.1 encodeTransformToCommandBuffer:sourceTexture:sourceHistogram:source↩ HistogramOffset:desiredHistogram:desiredHistogramOffset:(nonnull id< MTL↩ CommandBuffer > commandBuffer,[sourceTexture] nonnull id< MTLTexture > source,[sourceHistogram] nonnull id< MTLBuffer > sourceHistogram,[source↩ HistogramOffset] NSUInteger sourceHistogramOffset,[desiredHistogram] non-null id< MTLBuffer > desiredHistogram,[desiredHistogramOffset] NSUInteger desiredHistogramOffset) . . . 78

4.34.2.2 initWithDevice:histogramInfo:(nonnull id< MTLDevice > device,[histogramInfo] const MPSImageHistogramInfo *__nonnull NS_DESIGNATED_INITIALIZER) . . 79

4.34.3 Property Documentation . . . 79

4.34.3.1 histogramInfo . . . 79

4.35 MPSImageIntegral Class Reference . . . 80

4.35.1 Detailed Description . . . 80
4.36 MPSImageIntegralOfSquares Class Reference . . . 80
4.36.1 Detailed Description . . . 81
4.37 MPSImageLanczosScale Class Reference . . . 81
4.37.1 Detailed Description . . . 81
4.37.2 Property Documentation . . . 82
4.37.2.1 scaleTransform . . . 82
4.38 MPSImageLaplacian Class Reference . . . 82
4.38.1 Detailed Description . . . 83
4.38.2 Property Documentation . . . 83
4.38.2.1 bias . . . 83
4.39 MPSImageMedian Class Reference . . . 83
4.39.1 Detailed Description . . . 84
4.39.2 Method Documentation . . . 84
4.39.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 84
4.39.2.2 initWithDevice:kernelDiameter:(nonnull id< MTLDevice > device,[kernel↩ Diameter] NSUInteger NS_DESIGNATED_INITIALIZER) . . . 84
4.39.2.3 maxKernelDiameter() . . . 85
4.39.2.4 minKernelDiameter() . . . 85
4.39.3 Property Documentation . . . 85
4.39.3.1 kernelDiameter . . . 85
4.40 MPSImagePyramid Class Reference . . . 85
4.40.1 Detailed Description . . . 86
4.40.2 Method Documentation . . . 86
4.40.2.1 initWithDevice:(nonnull id< MTLDevice > device) . . . 86
4.40.2.2 initWithDevice:centerWeight:(nonnull id< MTLDevice > device,[centerWeight] float centerWeight) . . . 86
4.40.2.3 initWithDevice:kernelWidth:kernelHeight:weights:(nonnull id< MTLDevice > device,[kernelWidth] NSUInteger kernelWidth,[kernelHeight] NSUInteger kernelHeight,[weights] const float *__nonnull NS_DESIGNATED_INITIALIZ↩ ER) . . . 86
4.40.3 Property Documentation . . . 87
4.40.3.1 kernelHeight . . . 87
4.40.3.2 kernelWidth . . . 87
4.41 MPSImageSobel Class Reference . . . 87
4.41.1 Detailed Description . . . 88
4.41.2 Method Documentation . . . 88
4.41.2.1 initWithDevice:(nonnull id< MTLDevice > device) . . . 88
4.41.2.2 initWithDevice:linearGrayColorTransform:(nonnull id< MTLDevice > device,[linear↩ GrayColorTransform] const float *__nonnull NS_DESIGNATED_INITIALIZER) . . . 88
4.41.3 Property Documentation . . . 88
4.41.3.1 colorTransform . . . 88

4.42 MPSImageTent Class Reference . . . 89
4.42.1 Detailed Description . . . 89
4.43 MPSImageThresholdBinary Class Reference . . . 89
4.43.1 Detailed Description . . . 90
4.43.2 Method Documentation . . . 90
4.43.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 90
4.43.2.2 initWithDevice:thresholdValue:maximumValue:linearGrayColorTransform↩ :(nonnull id< MTLDevice > device,[thresholdValue] float thresholdValue,[maximum↩ Value] float maximumValue,[linearGrayColorTransform] const float *__nullable NS_DESIGNATED_INITIALIZER) . . . 91
4.43.3 Property Documentation . . . 92
4.43.3.1 maximumValue . . . 92
4.43.3.2 thresholdValue . . . 92
4.43.3.3 transform . . . 92
4.44 MPSImageThresholdBinaryInverse Class Reference . . . 92
4.44.1 Detailed Description . . . 93
4.44.2 Method Documentation . . . 93
4.44.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 93
4.44.2.2 initWithDevice:thresholdValue:maximumValue:linearGrayColorTransform↩ :(nonnull id< MTLDevice > device,[thresholdValue] float thresholdValue,[maximum↩ Value] float maximumValue,[linearGrayColorTransform] const float *__nullable NS_DESIGNATED_INITIALIZER) . . . 93
4.44.3 Property Documentation . . . 93
4.44.3.1 maximumValue . . . 93
4.44.3.2 thresholdValue . . . 93
4.44.3.3 transform . . . 93
4.45 MPSImageThresholdToZero Class Reference . . . 94
4.45.1 Detailed Description . . . 94
4.45.2 Method Documentation . . . 94
4.45.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 94
4.45.2.2 initWithDevice:thresholdValue:linearGrayColorTransform:(nonnull id< MT↩ LDevice > device,[thresholdValue] float thresholdValue,[linearGrayColor↩ Transform] const float *__nullable NS_DESIGNATED_INITIALIZER) . . . 95
4.45.3 Property Documentation . . . 95
4.45.3.1 thresholdValue . . . 95
4.45.3.2 transform . . . 95
4.46 MPSImageThresholdToZeroInverse Class Reference . . . 95
4.46.1 Detailed Description . . . 96
4.46.2 Method Documentation . . . 96
4.46.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 96
4.46.2.2 initWithDevice:thresholdValue:linearGrayColorTransform:(nonnull id< MT↩ LDevice > device,[thresholdValue] float thresholdValue,[linearGrayColor↩ Transform] const float *__nullable NS_DESIGNATED_INITIALIZER) . . . 96

4.46.3 Property Documentation . . . 96
4.46.3.1 thresholdValue . . . 96
4.46.3.2 transform . . . 96
4.47 MPSImageThresholdTruncate Class Reference . . . 97
4.47.1 Detailed Description . . . 97
4.47.2 Method Documentation . . . 97
4.47.2.1 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 97
4.47.2.2 initWithDevice:thresholdValue:linearGrayColorTransform:(nonnull id< MT↩ LDevice > device,[thresholdValue] float thresholdValue,[linearGrayColor↩ Transform] const float *__nullable NS_DESIGNATED_INITIALIZER) . . . 98
4.47.3 Property Documentation . . . 99
4.47.3.1 thresholdValue . . . 99
4.47.3.2 transform . . . 99
4.48 MPSImageTranspose Class Reference . . . 99
4.48.1 Detailed Description . . . 99
4.49 MPSKernel Class Reference . . . 100
4.49.1 Detailed Description . . . 100
4.49.2 Method Documentation . . . 101
4.49.2.1 copyWithZone:device:(nullable NSZone *zone,[device] nullable id< MTLDevice > device) . . . 101
4.49.2.2 initWithDevice:(nonnull id< MTLDevice > NS_DESIGNATED_INITIALIZER) . . . 102
4.49.2.3 MPSCopyAllocator . . . 103
4.49.3 Member Data Documentation . . . 104
4.49.3.1 MPSRectNoClip . . . 104
4.49.4 Property Documentation . . . 104
4.49.4.1 device . . . 104
4.49.4.2 label . . . 104
4.49.4.3 options . . . 104
4.50 MPSMatrix Class Reference . . . 104
4.50.1 Detailed Description . . . 105
4.50.2 Method Documentation . . . 105
4.50.2.1 initWithBuffer:descriptor:(nonnull id< MTLBuffer > buffer,[descriptor] nonnull M↩ PSMatrixDescriptor *descriptor) . . . 105
4.50.3 Property Documentation . . . 106
4.50.3.1 columns . . . 106
4.50.3.2 data . . . 106
4.50.3.3 dataType . . . 106
4.50.3.4 device . . . 106
4.50.3.5 rowBytes . . . 106
4.50.3.6 rows . . . 106
4.51 MPSMatrixDescriptor Class Reference . . . 106

4.51.1 Detailed Description . . . 107
4.51.2 Method Documentation . . . 107
4.51.2.1 matrixDescriptorWithDimensions:columns:rowBytes:dataType:(NSUInteger rows,[columns] NSUInteger columns,[rowBytes] NSUInteger rowBytes,[dataType] MPSDataType dataType) . . . 107
4.51.2.2 rowBytesFromColumns:dataType:(NSUInteger columns,[dataType] MPSDataType dataType) . . . 107
4.51.3 Property Documentation . . . 107
4.51.3.1 columns . . . 107
4.51.3.2 dataType . . . 107
4.51.3.3 rowBytes . . . 107
4.51.3.4 rows . . . 108
4.52 MPSMatrixMultiplication Class Reference . . . 108
4.52.1 Detailed Description . . . 108
4.52.2 Method Documentation . . . 109
4.52.2.1 encodeToCommandBuffer:leftMatrix:rightMatrix:resultMatrix:(nonnull id< MTLCommandBuffer > commandBuffer,[leftMatrix] MPSMatrix const *__nonnull leftMatrix,[rightMatrix] MPSMatrix const *__nonnull rightMatrix,[resultMatrix] encode(commandBuffer:leftMatrix:rightMatrix:resultMatrix:) MPS_SWIFT_NAME) . . . 109
4.52.2.2 initWithDevice:(nonnull id< MTLDevice > NS_UNAVAILABLE) . . . 109
4.52.2.3 initWithDevice:transposeLeft:transposeRight:resultRows:resultColumns:interiorColumns:alpha:beta:(nonnull id< MTLDevice > device,[transposeLeft] BOOL transposeLeft,[transposeRight] BOOL transposeRight,[resultRows] NSUInteger resultRows,[resultColumns] NSUInteger resultColumns,[interiorColumns] NSUInteger interiorColumns,[alpha] double alpha,[beta] double beta) . . . 109
4.52.3 Property Documentation . . . 110
4.52.3.1 leftMatrixOrigin . . . 110
4.52.3.2 resultMatrixOrigin . . . 110
4.52.3.3 rightMatrixOrigin . . . 110
4.53 MPSOffset Struct Reference . . . 110
4.53.1 Detailed Description . . . 110
4.53.2 Member Data Documentation . . . 111
4.53.2.1 x . . . 111
4.53.2.2 y . . . 111
4.53.2.3 z . . . 111
4.54 MPSOrigin Struct Reference . . . 111
4.54.1 Detailed Description . . . 111
4.54.2 Member Data Documentation . . . 111
4.54.2.1 x . . . 111
4.54.2.2 y . . . 111
4.54.2.3 z . . . 111
4.55 MPSRegion Struct Reference . . . 112

4.55.1 Detailed Description . . . 112
4.55.2 Member Data Documentation . . . 112
4.55.2.1 origin . . . 112
4.55.2.2 size . . . 112
4.56 MPSScaleTransform Struct Reference . . . 112
4.56.1 Detailed Description . . . 112
4.57 MPSSize Struct Reference . . . 113
4.57.1 Detailed Description . . . 113
4.57.2 Member Data Documentation . . . 113
4.57.2.1 depth . . . 113
4.57.2.2 height . . . 113
4.57.2.3 width . . . 113
4.58 MPSTemporaryImage Class Reference . . . 113
4.58.1 Detailed Description . . . 114
4.58.2 Method Documentation . . . 114
4.58.2.1 initWithDevice:imageDescriptor:(nonnull id< MTLDevice > device,[image↩ Descriptor] const MPSImageDescriptor *__nonnull NS_UNAVAILABLE) . . . 114
4.58.2.2 initWithTexture:featureChannels:(nonnull id< MTLTexture > texture,[feature↩ Channels] NSUInteger NS_UNAVAILABLE) . . . 115
4.58.2.3 prefetchStorageWithCommandBuffer:imageDescriptorList:(nonnull id< MTL↩ CommandBuffer > commandBuffer,[imageDescriptorList] NSArray< MPS↩ ImageDescriptor * > *__nonnull descriptorList) . . . 115
4.58.2.4 temporaryImageWithCommandBuffer:imageDescriptor:(nonnull id< MTL↩ CommandBuffer > commandBuffer,[imageDescriptor] const MPSImage↩ Descriptor *__nonnull imageDescriptor) . . . 115
4.58.2.5 temporaryImageWithCommandBuffer:textureDescriptor:(nonnull id< MTL↩ CommandBuffer > commandBuffer,[textureDescriptor] const MTLTexture↩ Descriptor *__nonnull textureDescriptor) . . . 115
4.59 MPSUnaryImageKernel Class Reference . . . 116
4.59.1 Detailed Description . . . 117
4.59.2 Method Documentation . . . 117
4.59.2.1 encodeToCommandBuffer:inPlaceTexture:fallbackCopyAllocator:(nonnull id< MTLCommandBuffer > commandBuffer,[inPlaceTexture] __nonnull id< MTL↩ Texture > __strong *__nonnull texture,[fallbackCopyAllocator] encode(command↩ Buffer:inPlaceTexture:fallbackCopyAllocator:) MPS_SWIFT_NAME) . . . 117
4.59.2.2 encodeToCommandBuffer:sourceTexture:destinationTexture:(nonnull id< MT↩ LCommandBuffer > commandBuffer,[sourceTexture] nonnull id< MTLTexture > sourceTexture,[destinationTexture] encode(commandBuffer:sourceTexture↩ :destinationTexture:) MPS_SWIFT_NAME) . . . 118
4.59.2.3 sourceRegionForDestinationSize:(sourceRegion(destinationSize:) MPS_SWIF↩ T_NAME) . . . 118
4.59.3 Property Documentation . . . 119
4.59.3.1 clipRect . . . 119
4.59.3.2 edgeMode . . . 119

4.59.3.3 offset 119

Index 121

Chapter 1

Metal Shaders - High Performance Kernels on Metal

MetalPerformanceShaders.h MetalPerformanceShaders

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved.

1.1 Introduction

MetalPerformanceShaders.framework is a framework of highly optimized compute and graphics shaders that are designed to integrate easily and efficiently into your Metal application. These data-parallel primitives are specially tuned to take advantage of the unique hardware characteristics of each iOS GPU to ensure optimal performance. Applications adopting MetalPerformanceShaders can be sure of achieving optimal performance without needing to update their own hand-written shaders for each new iOS GPU generation. MetalPerformanceShaders can be used along with the application's existing Metal resources (such as the MTLCommandBuffer, MTLBuffer and MTLTexture objects) and shaders.

In iOS 9, MetalPerformanceShaders.framework provides a series of commonly-used image processing primitives for image effects on Metal textures.

In iOS 10, MetalPerformanceShaders.framework adds support for the following kernels:

- collection of kernels to implement and run neural networks using previously obtained training data, on the GPU
- new image processing filters to perform color-conversion and for building a gaussian pyramid

1.2 Data containers

1.2.1 MTLTextures and MTLBuffers

Most data operated on by Metal Performance Shaders must be in a portable data container appropriate for use on the GPU, such as a MTLTexture, MTLBuffer or MPSImage. The first two should be self-explanatory based on your previous experience with Metal.framework. MPS will use these directly when it can.

1.2.2 MPSImages

Convolutional neural networking (CNN) filters may need more than the four data channels that a MTLTexture can provide. In these cases, the MPSImage is used instead as an abstraction layer on top of a MTLTexture. When more than 4 channels are needed, additional textures in the texture2d array are added to hold additional channels in sets of four. The MPSImage tracks this information as the number of "feature channels" in an image.

1.2.3 MPSTemporaryImages

The MPSTemporaryImage (subclass of MPSImage) extends the MPSImage to provide advanced caching of unused memory to increase performance and reduce memory footprint. They are intended as fast GPU-only storage for intermediate image data needed only transiently within a single MTLCommandBuffer. They accelerate the common case of image data which is created only to be consumed and destroyed immediately by the next operation(s) in a MTLCommandBuffer. MPSTemporaryImages provide convenient and simple way to save memory by automatically aliasing other MPSTemporaryImages in the MTLCommandBuffer. Because they alias (share texel storage with) other textures in the same MTLCommandBuffer, the valid lifetime of the data in a MPSTeporaryImage is extremely short, limited to a portion of a MTLCommandBuffer. You can not read or write data to a MPSTemporaryImage using the CPU, or use the data in other MTLCommandBuffers. Use regular MPSImages for more persistent storage.

1.3 The MPSKernel

The MPSKernel is the base class for all MPS kernels. It defines baseline behavior for all MPS kernels, declaring the device to run the kernel on, some debugging options and a user-friendly label, should one be required. From this are derived the MPSUnaryImageKernel and MPSBinaryImageKernel sub-classes which define shared behavior for most image processing kernels (filters) such as edging modes, clipping and tiling support for image operations that consume one or two source textures. Neither these or the MPSKernel are typically be used directly. They just provide API abstraction and in some cases may allow some level of polymorphic manipulation of MPS image kernel objects.

Subclasses of the MPSUnaryImageKernel and MPSBinaryImageKernel provide specialized -init and -encode methods to encode various image processing primitives into your MTLCommandBuffer, and may also provide additional configurable properties on their own. Many such image filters are available: There are convolutions (generic, box, Sobel, and Gaussian) to do edge detection, sharpening and blurring, morphological operators – Min, Max, Dilate and Erode – and histogram operations. In addition, there are median, resampling filters and others. All of these run on the GPU directly on MTLTextures and MTLBuffers.

As the MPSKernel/MPSUnaryImageKernel/MPSBinaryImageKernel classes serve to unify a diversity of image operations into a simple consistent interface and calling sequence to apply image filters, subclasses implement details that diverge from the norm. For example, some filters may take a small set of parameters (e.g. a convolution kernel) to govern how they function. However, the overall sequence for using MPSKernel subclasses remains the same:

1. Allocate the usual Metal objects: MTLDevice, MTLCommandQueue, and MTLCommandBuffer to drive a Metal compute pipeline. If your application already uses Metal, chances are you have most of these things already. MPS will fit right in to this workflow. It can encode onto MTLCommandBuffers inline with your own workload.

2. Create an appropriate MPSKernel object. For example, if you want to do a Gaussian blur, make a MPSImageGaussianBlur object. MPSKernel objects are generally light weight but can be reused to save some setup time. They can not be used by multiple threads concurrently, so if you are using Metal from many threads concurrently, make extra MPSKernel objects. MPSKernel objects conform to <NSCopying>.

3. Call [MPSKernelSubclass encodeToCommandBuffer:...]. Parameters for other -encode... calls vary by filter type, but operate similarly. They create a MTLCommandEncoder, write commands to run the filter into the MTLCommandBuffer and then end the MTLCommandEncoder. This means you must call -endEncoding on your current MTLCommandEncoder before calling a MPSKernel encode method. You can at this point release the MPSKernel or keep it around to use again to save some setup cost.

4. If you wish to encode futher commands of your own on the MTLCommandBuffer, you must create a new MTLCommandEncoder to do so.

5. (Standard Metal) When you are done with the MTLCommandBuffer, submit it to the device using typical Metal commands, such as [MTLCommandBuffer commit]. The MPS filter will begin running on the GPU. You can either use [MTLCommandBuffer waitUntilCompleted] or [MTLCommandBuffer addCompletedHandler:] to be notified when the work is done.

Each MPSKernel is allocated against a particular MTLDevice. A single filter may not be used with multiple MTLDevices. (You will need to make multiple MPSKernels for that.) This is necessary because the [MPSKernel initWithDevice:...] methods sometimes allocate MTLBuffers and MTLTextures to hold data passed in as parameters to the -init method and a MTLDevice is required to make those. MPSKernels provide a copy method that allow them to be copied for a new device.

MPSKernel objects are not entirely thread safe. While they may be used in a multithreaded context, you should not attempt to have multiple MPSKernel objects writing to the same MTLCommandBuffer at the same time. They share restrictions with the MTLCommandEncoder in this regard. In limited circumstances, the same MPSKernel can be used to write to multiple MTLCommandBuffers concurrently. However, that only works if the MPSKernel is treated as an immutable object. That is, if MPSKernel subclass properties of a shared filter are changed, then the change can be reflected on the other thread while the other thread is encoding its work, leading to undefined behavior. It is generally safest to just make copies of MPSKernel objects, one for each thread.

For more information, please see MPSTypes.h.

1.3.1 MPS{Unary/Binary}ImageKernel properties

The MPS{Unary/Binary}ImageKernel base classes define several properties common to all MPSKernels:

1.3.1.1 MPSKernel clipRect

The clipRect property, common to MPSKernel sublcasses that write to a destination texture, describes the sub-rectangle of the destination texture overwritten by the filter. If the clipRect is larger than the destination texture, the intersection between the clipRect and destination texture bounds will be used. The clipRect may be used to avoid doing work to obscured regions of the destination image, or to manage tiling and to limit operations to parts of an image if for example, the user drew a rectangle on the screen and asked you to just apply the filter there.

```
extern MTLRegion MPSRectNoClip; //Pass this rectangle to fill the entire destination texture.
```

1.3.1.2 MPSOffset

The offset (or primaryOffset or secondaryOffset) property, common to MPSKernel subclasses that use a source texture from which pixel data is read, describes the positioning of the source image relative to the result texture. A offset of {0,0,0} indicates that the top left pixel of the source texture is the center pixel used to create the top left corner of the destination texture clipRect. An offset of {1,2,0} positions the top left corner of the clipRect at {x=1, y=2, z=0} of the source image. The offset is the position of the top left corner of the clipRect in the source coordinate frame. It can be used for tiling and for translating an image up/down or left/right by pixel increments. If there is no clipRect then the offset is the top left corner of the region read by the filter. If there are multiple source textures, then the primaryOffset describes the top left corner of the region read in the primary source texture. The secondaryOffset describes the top left corner of the region read in the secondary source texture, and so forth.

1.3.1.3 MPSKernelEdgeMode

The edgeMode (or primaryEdgeMode or secondaryEdgeMode)describes the behavior of texture reads that stray off the edge of the source image. This can happen if the offset is negative, meaning read off the top or left edge of the image. It can also happen if the clipRect.size + offset is larger than the source image, meaning read off the bottom and right of the image. It is also possible for filters that have a filter window that stretches to examine neighboring pixels, such as convolution, morphology and resampling filters. If there are multiple source textures, then the primaryEdgeMode describes the MPSKernelEdgeMode to use with primary source texture. The secondaryEdgeMode describes the MPSKernelEdgeMode to use with the secondary source texture, and so forth.

```
typedef NS_ENUM(NSUInteger, MPSImageEdgeMode)

MPSImageEdgeModeZero        Out of bound pixels are (0,0,0,1) for image formats without
                                 alpha channel and (0,0,0,0) for image with pixel format with an
                                 alpha channel

MPSImageEdgeModeClamp       Out of bound pixels are clamped to nearest edge pixel
```

1.3.1.4 MPSKernelOptions

Each MPSKernel takes a MPSKernelOptions bit mask to indicate various options to use when running the filter:

```
typedef NS_OPTIONS(NSUInteger, MPSKernelOptions)

MPSKernelOptionsNone                      Use default options

MPSKernelOptionsSkipAPIValidation         Do not spend time looking at parameters passed to MPS
                                          for errors.

MPSKernelOptionsAllowReducedPrecision     When possible, MPSKernels use a higher precision data representation
                                          the destination storage format to avoid excessive accumulation of comp
                                          rounding error in the result. MPSKernelOptionsAllowReducedPrecision ac
                                          MPSKernel that the destination storage format already has too much pre
                                          what is ultimately required downstream, and the MPSKernel may use redu
                                          internally when it feels that a less precise result would yield better
                                          When enabled, the precision of the result may vary by hardware and ope
```

1.4 Available MPSKernels

1.4.1 Image Convolution

1.4.1.1 The Image Convolution Kernel

The convolution filter is at its simplest the weighted average of a pixel with its nearest neighbors. The weights are provided by a convolution kernel. The number and position of the nearest neighbors that are considered are given by the size of the convolution kernel. For example, a convolution kernel might be the following 5x5 array of weights:

```
1       2       3       2       1
2       4       6       4       2
3       6      [9]      6       3
2       4       6       4       2
1       2       3       2       1
```

In order to calculate this 5x5 convolution result, one would multiply all of the pixels in the image within (5/2=) 2 pixels of the desired pixel by its corresponding weight, add them up and divide by a divisor to renormalize the results. Then, repeat for all other pixels in the area you wish to convolve.

For those MPS filters where the convolution kernel is passed in, you provide the kernel as a normalized float array. That is, the kernel weights have the divisor already divided into them and as a consequence should usually sum to 1.0. In our tent example above, the sum over the area of the kernel is 81, so one would normalize it as follows:

```
1/81    2/81    3/81    2/81    1/81
2/81    4/81    6/81    4/81    2/81
3/81    6/81   [9/81]   6/81    3/81
2/81    4/81    6/81    4/81    2/81
1/81    2/81    3/81    2/81    1/81
```

It is not strictly necessary that the filter weights add up to 1.0f. Edge detection filters frequently add up to zero. You may decide to have the area under the filter be a bit bigger or smaller than 1.0 to increase or reduce the contrast in the result.

The MxN kernel is passed in as a 1-dimensional data array in row major order.

Some convolution filters also have a notion of a bias. This is a number to be added to the result before it is written out to result texture. A bias might be used to uniformly brighten an image, set a video range baseline (e.g. 0 might actually be encoded as 16/255) or to make negative signal representable on a unorm image.

```
A unorm image is an image comprised of unsigned normalized samples. A typical 8-bit image (e.g.
MTLPixelFormatRGBA8Unorm) is a unorm image. It has unsigned samples that represent values between
[0,1]. In the case of MTLPixelFormatRGBA8Unorm, the encoding of 0 is 0, and the encoding of 1.0f
is UINT8_MAX (255).
```

1.4.1.2 The Box, Tent and Gaussian Filters

There are many different convolution kernel shapes which can produce different results. A kernel consisting of all 1's is called a Box filter. It is very quick to calculate and may run nearly as fast as a texture copy, even for very large blur radii. The blur effect that you get, however, can be square in appearance and may not be entirely appealing under close scrutiny. A second pass of the box will lead to a Tent kernel. (The 5x5 tent above can be reduced into two 3x3 Box filters.) Its appearance is more pleasing. Tent operations can be found in sample code for window shadows. Both Box and Tent filters are provided by MPS. Multiple passes of a box and/or tent filters will tend to converge towards a gaussian line shape and produce a smoother blur effect. MPS also provides a Gaussian blur, though it uses a different method.

1.4.1.3 Laplacian and Unsharp Mask Filters

One can in practice also subtract a blur from the image to produce a sharpening effect (unsharp mask). This is done by preparing a convolution kernel which is a scaled image less a blur to reduce the low frequency component of an image. This can reduce blur, but may also emphasize noise in the image. As an example, we can do identity minus a box blur:

```
             | 1   1   1 |
k0 * [1] -   | 1   1   1 | * k2
             | 1   1   1 |
```

If we pick k0 = 9 and k2 = 1, so that the two kernels have equal weight, we arrive at:

```
|-1  -1  -1 |
|-1   8  -1 |
|-1  -1  -1 |
```

This is a Laplacian filter for calculating image gradients (including diagonals in this case).

```
Caution: because this convolution kernel has negative regions, it can produce negative
results as well as positive ones from ordinary image data. If you intend to store the
result in a unorm texture, you'll need to scale it and add a positive value to it to avoid
having the negative signal clamped off. (e.g. p' = 0.5*p+0.5).
```

An unsharp mask filter is the sum between the original image and a scaled result of the Laplacian filter. The scaling factor (and filter size and shape) adjusts the nature of the low frequency signal and the degree to which it is removed. This work can usually be combined into the convolution kernel, to do the whole thing in one pass.

1.4.1.4 Sobel Edge detection

Instead of integrating over an area, Convolution filters can also differentiate over an area by subtracting adjacent pixels. One such filter is the Sobel edge detection filter. It produces bright signal where there are large differences between one pixel and the next and black elsewhere:

```
   | -1   0   1 |        | 1   2   1 |
Gx=| -2   0   2 |   Gy= | 0   0   0 |
   | -1   0   1 |        |-1  -2  -1 |


   result = sqrt( Convolve(src, Gx) * Convolve(src * Gx) +
                  Convolve(src, Gy) * Convolve(src * Gy))
```

1.4.1.5 Other Filters

Other effects can be achieved as well, such as emboss:

```
-1   0   0
 0   0   0
 0   0   1
```

1.4.1.6 Separable Convolution

Some convolution kernels are separable. That is, the filter weights can be factored into the product of two smaller sets of weights. As an example, the tent kernel shown above can be factored into a horizontal and vertical 1-dimensional kernel each containing [1 2 3 2 1]. In this way, what might otherwise have been a 5x5 convolution with 25 multiplies and 24 adds is instead a 5x1 and 1x5 convolution with a total of 10 multiplies and 8 adds and possibly some extra load/store traffic for the two-pass algorithm. The savings get bigger for bigger filter areas. MPS convolution filters will automatically separate kernels to run faster, when possible. Some filters with fixed kernels such as Box and Guassian are inherently separable. We attempt to factor the general convolution kernel into 2 1D kernels in the -initWithDevice:... method. If you want to factor it yourself, make two MPSImageConvolution objects with 1D kernels.

1.4.1.7 Convolutions in MPS

Convolution filters provided by MPS include:

```
MPSImageConvolution       <MetalPerformanceShaders/MPSImageConvolution.h>       General convolution
MPSImageGassianBlur       <MetalPerformanceShaders/MPSImageConvolution.h>       Gaussian blur
MPSImageBox               <MetalPerformanceShaders/MPSImageConvolution.h>       Box blur
MPSImageTent              <MetalPerformanceShaders/MPSImageConvolution.h>       Tent blur
```

1.4.2 Morphology

Morphological operators are similar to convolutions in that they find a result by looking at the nearest neighbors of each pixel in the image. Instead of calculating a weighted average, morphological operators scan the kernel area looking for the maximum or minimum pixel value. The MPSImageAreaMax and MPSImageAreaMin filters return the raw maximum and minimum color channel value in the kernel area for each pixel, respectively. The MPSImageDilate and MPSImageErode filters do the same thing, except that the probe shape need not be a rectangle, and instead can be nearly any shape you desire, such as a antialiased oval, star or heart shape.

When applied, the max and dilate filters have the effect of adding their shape on to the periphery of bright objects in the image. A single bright pixel, such as might be found in a photograph of a starry night sky will become the shape of a probe – a rectangle for max, and perhaps a 5-pointed star if that is the shape you chose for the dilate filter kernel. Larger objects will adopt more rectangular or star quality into their shape. (An oval or circular probe would round the corners of a rectangular object, for example.) The min and erode filters do similar things to the darker regions of the image.

When a dilate filter is followed by an erode filter (or max followed by min) with similar filters, the effect is known as a close operator. Expanding bright areas only to erode them away again leaves most of the image in roughly the same shape as it started, but small dark areas that are completely removed by the dilate operator are not replaced by the erode. Dark noise may be removed. Small enclosed dark area may be completely filled in by bright signal. Similarly erode followed by dilate is an open operator. It will tend to remove bright fine detail and fill in small bright areas surrounded by dark lines.

To make a MPS morphology filter with a text glyph draw black text on a white background. MPS morphology filters must have a center pixel with value 0.

Morphology filters provided by MPS include:

```
MPSImageAreaMax  <MetalPerformanceShaders/MPSImageMorphology.h>       Area Max
MPSImageAreaMin  <MetalPerformanceShaders/MPSImageMorphology.h>       Area Min
MPSImageDilate   <MetalPerformanceShaders/MPSImageMorphology.h>       Dilate
MPSImageErode    <MetalPerformanceShaders/MPSImageMorphology.h>       Erode
```

1.4.3 Histogram

A image may be examined by taking the histogram of its pixels. This gives the distribution of the various intensities per color channel. The MPSImageHistogram filter can be used to calculate a histogram for a MTLTexture.

In some cases, as a result of image processing operations the very dark and light regions of the intensity spectrum can become unpopulated. Perhaps a photograph is underexposed or overexposed. The MPSImageHistogramEqualization filter will redistribute the intensities to a more uniform distribution, correcting such problems. The MPSImageHistogramSpecification class allows you to cause an image to conform to a different histogram.

Histogram filters provided by MPS include:

```
MPSImageHistogram              <MetalPerformanceShaders/MPSImageHistogram.h>     Calculate the histogram of an
MPSImageHistogramEqualization  <MetalPerformanceShaders/MPSImageHistogram.h>     Redistribute intensity in an
                                                                                 the histogram
MPSImageHistogramSpecification <MetalPerformanceShaders/MPSImageHistogram.h>     A generalized version of hist
                                                                                 operation. Convert the image so that its h
                                                                                 matches the desired histogram provided to
                                                                                 histogram specification filter.
```

1.4.4 Image Median

Median filters find the median value in a region surrounding each pixel in the source image. It is frequently used to remove noise from the image, but may also be used to remove fine detail like a open filter. It is widely used in image processing because in many cases it can remove noise while at the same time preserving edges.

Median filters provided by MPS include:

```
MPSImageMedian                 <MetalPerformanceShaders/MPSImageMedian.h>         Calculate the median of an im
                                                                                  square filter window.
```

1.4.5 Image Resampling

Resampling operations are used to convert one regular array of pixels to another regular array of pixels, typically along a different set of axes and/or using a different sampling period. Changing the sampling period will enlarge or reduce images and/or distort the aspect ratio. Change of axis results in rotations or arbitrary affine transforms.

For most imaging work on the GPU, resampling can be quickly and simply done as part of another pass using a Euler matrices or quaternions to transform the coordinate space followed by linear filtering to interpolate the value found there. However, this can lead to somewhat muddy images and may result in loss of signal when downsampling by more than a factor of two unless a low pass filter is applied first. It is also prone to the development of Moire patterns in regions of the image with regularly repeating signal, such as a picture of a masonry grid on the side of a building.

The MPS resampling routines use a higher quality (but more expensive) Lanczos resampling algorithm. Especially with photographic images, it will usually produce a much nicer result. It does not require a low pass filter be applied to the image before down sampling. However, some ringing can occur near high frequency regions of the image, making the algorithm less suitable for vector art.

MetalPerformanceShaders.framework provides a MPSImageLanczosScale function to allow for simple resizing of images into the clipRect of the result image. It can operate with preservation of aspect ratio or not.

```
MPSImageLanczosScale           <MetalPerformanceShaders/MPSResample.h>   Resize or adjust aspect ratio of a
```

1.4.6 Image Threshold

Thresholding operations are commonly used to separate elements of image structure from the rest of an image. Generally, these operate by making some sort of simple comparison test, for example color_intensity $>$ 0.5, and then writing out 0 or 1 (actual values configurable) depending on the truth or falsehood of the result. It is frequently used in computer vision, or to accentuate edge detection filters.

A variety of thresholding operators are supported:

```
MPSImageThresholdBinary          <MetalPerformanceShaders/MPSImageThreshold.h>  srcPixel > thresholdVal ? max
MPSImageThresholdBinaryInverse   <MetalPerformanceShaders/MPSImageThreshold.h>  srcPixel > thresholdVal ? 0 :
MPSImageThresholdTruncate        <MetalPerformanceShaders/MPSImageThreshold.h>  srcPixel > thresholdVal ? thr
MPSImageThresholdToZero          <MetalPerformanceShaders/MPSImageThreshold.h>  srcPixel > thresholdVal ? src
MPSImageThresholdToZeroInverse   <MetalPerformanceShaders/MPSImageThreshold.h>  srcPixel > thresholdVal ? 0 :
```

1.4.7 Convolutional Neural Networks

Convolutional Neural Networks (CNN) is a machine learning technique that attempts to model the visual cortex as a sequence of convolution, rectification, pooling and normalization steps. Several CNN filters commonly derived from the MPSCNNKernel base class are provided to help you implement these steps as efficiently as possible.

```
MPSCNNNeuronLinear              <MetalPerformanceShaders/MPSCNN.h>      A linear neuron activation function
MPSCNNNeuronReLU                <MetalPerformanceShaders/MPSCNN.h>      A neuron activation function with rect
MPSCNNNeuronSigmoid             <MetalPerformanceShaders/MPSCNN.h>      A sigmoid neuron activation function 1
MPSCNNNeuronTanH                <MetalPerformanceShaders/MPSCNN.h>      A neuron activation function using hyp
MPSCNNConvolution               <MetalPerformanceShaders/MPSCNN.h>      A 4D convolution tensor
MPSCNNFullyConnected            <MetalPerformanceShaders/MPSCNN.h>      A fully connected CNN layer
MPSCNNPoolingMax                <MetalPerformanceShaders/MPSCNN.h>      The maximum value in the pooling area
MPSCNNPoolingAverage            <MetalPerformanceShaders/MPSCNN.h>      The average value in the pooling area
MPSCNNSpatialNormalization      <MetalPerformanceShaders/MPSCNN.h>
MPSCNNCrossChannelNormalization <MetalPerformanceShaders/MPSCNN.h>
MPSCNNSoftmax                   <MetalPerformanceShaders/MPSCNN.h>      exp(pixel(x,y,k))/sum(exp(pixel(x,y,0)
MPSCNNLogSoftmax                <MetalPerformanceShaders/MPSCNN.h>      pixel(x,y,k) - ln(sum(exp(pixel(x,y,0)
```

MPSCNNKernels operate on MPSImages. MPSImages are at their core MTLTextures. However, whereas MTLTextures commonly represent image or texel data, a MPSImage is a more abstract representation of image features. The channels within a MPSImage do not necessarily correspond to colors in a color space. (Though, they can.) As a result, there can be many more than four of them. 32 or 64 channels per pixel is not uncommon. This is achieved on the MTLTexture hardware abstraction by inserting extra RGBA pixels to handle the additional feature channels (if any) beyond 4. These extra pixels are stored as multiple slices of a 2D image array. Thus, each CNN pixel in a 32-channel image is represented as 8 array slices, with 4-channels stored per-pixel in each slice. The width and height of the MTLTexture is the same as the width and height of the MPSImage. The number of slices in the MTLTexture is given by the number of feature channels rounded up to a multiple of 4.

MPSImages can be created from existing MTLTextures. They may also be created anew from a MPSImageDescriptor and backed with either standard texture memory, or as MPSTemporaryImages using memory drawn from MPS's internal cached texture backing store. MPSTemporaryImages can provide great memory usage and CPU time savings, but come with significant restrictions that should be understood before using them. For example, their contents are only valid during the GPU-side execution of a single MTLCommandBuffer and can not be read from or written to by the CPU. They are provided as an efficient way to hold CNN computations that are used immediately within the scope of the same MTLCommandBuffer and then discarded. We also support concatenation by allowing the user to define from which destination feature channel to start writing the output of the current layer. In this way the application can make a large MPSImage or MPSTemporaryImage and fill in parts of it with multiple layers (as long as the destination feature channel offset is a multiple of 4).

Some CNN Tips:

- Think carefully about the edge mode requested for pooling layers. The default is clamp to zero, but there are times when clamp to edge value may be better.
- To avoid falling off the edge of an image for filters that have a filter area (convolution, pooling) set the MPSCNNKernel.offset = (MPSOffset){ .x = kernelWidth/2, .y = kernelHeight/2, .z = 0}; and reduce the size of the output image by {kernelWidth-1, kernelHeight-1,0}. The filter area stretcheds up and to the left of the MPSCNNKernel.offset by {kernelWidth/2, kernelHeight/2}. While consistent with other MPS imaging operations, this behavior is different from some other CNN implementations.
- Please remember: MPSCNNConvolution takes weights in the order weight[outputChannels][kernelHeight][kernelWidth][inputChannels / groups] MPSCNNFullyConnected takes weights in the order weight[outputChannels][sourceWidth][sourceHeight][inputChannels]
- Initialize MPSCNNKernels once and reuse them
- You can use MPSCNNNeurons and other Filters in MPS to perform pre-processing of images, such as scaling and resizing.
- Specify a neuron filter with MPSCNNConvolution descriptor to combine the convolution and neuron operations.
- Use MPSTemporaryImages for intermediate images that live for a short period of time (less than one MTLCommandBuffer). MPSTemporaryImages can reduce the amount of memory used by the convolutional neural

network by several fold, and similarly reduce the amount of CPU time spent allocating storage and latency between MTLCommandBuffer.commit and when the work actually starts on the GPU. MPSTemporaryImage are for short lived storage within the time period of the execution of a single MTLCommandBuffer. You can not read or write to a MPSTemporaryImage using the CPU. Generally, they should be created as needed and thrown away promptly. Persistent objects should not retain them. Please be sure to understand the use of the MPSTemporaryImage.readCount.

- Because MPS encodes its work in place in your MTLCommandBuffer, you always have the option to insert your own code in between MPSCNNKernels as a Metal shader for tasks not covered by MPS. You need not use MPS for everything.

1.5 MPS API validation

MPS uses the same API validation layer that Metal uses to alert you to API mistakes while you are developing your code. While this option is turned on (Xcode: Edit Scheme: options: Metal API Validation), common programming errors will either trigger an assert or send a warning to the the debug log. Except in the case of serious errors, little or no spew should arrive in the console under standard usage. You can also try the MPSKernel.options parameter MPSKernelOptionsSkipAPIValidation to skip most of this checking. The flag may also lead to small reductions in CPU cost.

Note: where APIs are tagged nonnull, MPS expects that the value is not NULL. The validation layer may do some checking and assert. If you turn that off, then undefined behavior is the result of passing nil, and your application will likely be terminated.

1.6 How to Add MetalPerformanceShaders.framework to your project

Xcode:

```
1.  Click project file at left, then appropriate target, then select Build Phases.
2.  Open the "Link Binary With Libraries" disclosure triangle
3.  Click the [+] button in the "Link Binary With Libraries" view to add a new library
4.  Select MetalPerformanceShaders.framework from the list.
5.  Click the Add button.
```

Command Line:

```
clang  -framework MetalPerformanceShaders    file.c -o file.o
```

1.7 How to Determine if MPS Works on Your Device

To test whether MPS works on your device, you may call MPSSupportsMTLDevice(id<MTLDevice>). It will return YES if the device is supported.

1.8 In Place Operation

Some MPS filters can operate in place. In-place operation means that the same texture is used to hold both the input image and the result image. Operating in place is a great way to save memory, time and energy. You can use a MPS filter in place using [MPSKernel encodeToCommandBuffer:inPlaceTexture:copyAllocator:].

Unfortunately, it is not always possible for MPS filters to run in place. Whether a particular MPSKernel can operate in place can vary according to the hardware it is running on, the operating system version and the parameters and properties passed to it. You may not assume that because a MPSKernel works in place today on a particular device that it will continue to do so in the future.

To simplify error handling with failed in-place operation, [MPSKernel encodeToCommandBuffer:inPlaceTexture↩ :fallbackCopyAllocator:] takes an optional MPSCopyAllocator parameter. It is used to create a new texture when in-place operation is not possible so as to allow the operation to proceed out of place in a reliable fashion instead. (When this happens the input texture is released and replaced with a new texture.) To make use of this feature, you will need to write a MPSCopyAllocator block.

1.8.1 MPSCopyAllocator

Some MPSKernel objects may not be able to operate in place. When that occurs, and in-place operation is requested, MPS will call back to this block to get a new texture to overwrite instead. To avoid spending long periods of time allocating pages to back the MTLTexture, the block should attempt to reuse textures. The texture returned from the MPSCopyAllocator will be returned instead of the sourceTexture from the MPSKernel method on return. Here is a minimal MPSCopyAllocator implementation:

```
// A MPSCopyAllocator to handle cases where in-place operation fails.
MPSCopyAllocator myAllocator = ^id <MTLTexture>( MPSKernel * __nonnull filter,
                                                 __nonnull id <MTLCommandBuffer> cmdBuf,
                                                 __nonnull id <MTLTexture> sourceTexture)
{
    MTLPixelFormat format = sourceTexture.pixelFormat;  // FIXME: is this format writable?
    MTLTextureDescriptor *d = [MTLTextureDescriptor texture2DDescriptorWithPixelFormat: format
                                  width: sourceTexture.width
                                 height: sourceTexture.height
                              mipmapped: NO];
    d.usage = MTLTextureUsageShaderRead | MTLTextureUsageShaderWrite;

    //FIXME: Allocating a new texture each time is slow. They take up to 1 ms each.
    //       There are not too many milliseconds in a video frame! You can recycle
    //       old textures (or MTLBuffers and make textures from them) and reuse
    //       the memory here.
    id <MTLTexture> result = [cmdBuf.device newTextureWithDescriptor: d];

    // FIXME: If there is any metadata associated with sourceTexture such as colorspace
    //        information, MTLResource.label, MTLResource.cpuCacheMode mode,
    //        MTLResource.MTLPurgeableState, etc., it may need to be similarly associated
    //        with the new texture, to avoid losing your metadata.

    // FIXME: If filter.clipRect doesn't cover the entire image, you may need to copy
    //        pixels from sourceTexture to result or regions of result will be
    //        uninitialized. You can make a MTLCommandEncoder to encode work on the
    //        MTLCommandBuffer here to do that work, if necessary. It will be scheduled
    //        to run immediately before the MPSKernel work. Do not call
    //        [MTLCommandBuffer enqueue/commit/waitUntilCompleted/waitUntilScheduled]
    //        in the MPSCopyAllocator block. Make sure to call -endEncoding on the
    //        MTLCommandEncoder so that the MTLCommandBuffer has no active encoder
    //        before returning.

    // CAUTION: The next command placed on the MTLCommandBuffer after the MPSCopyAllocator
    //          returns is almost assuredly going to be encoded with a MTLComputeCommandEncoder.
    //          Creating any other type of encoder in the MPSCopyAllocator will probably cost
    //          an additional 0.5 ms of both CPU _AND_ GPU time (or more!) due to a double
    //          mode switch penalty.

    return result;
    // d is autoreleased
};
```

filter	A valid pointer to the MPSKernel that is calling the MPSCopyAllocator. From it you can get the clipRect of the intended operation.
cmdBuf	A valid MTLCommandBuffer. It can be used to obtain the device against which to allocate the new texture. You may also enqueue operations on the commandBuffer to initialize the texture. You may not submit, enqueue or wait for completion of the command buffer.
sourceTexture	The texture that is providing the source image for the filter. You may wish to copy its size and MTLPixelFormat for the new texture, but it is not requred.
return	A new valid MTLTexture to use as the destination for the MPSKernel. The format of the returned texture does not need to match sourceTexture.

1.9 Sample Code

```
#import <MetalPerformanceShaders/MetalPerformanceShaders.h>
```

```
// Blur the input texture (in place if possible) on MTLCommandQueue q, and return the new texture.
// This is a trivial example. It is not necessary or necessarily advised to enqueue a MPSKernel on
// its own MTLCommandBuffer  or using its own MTLComputeCommandEncoder. Group work together.
//
// Here we assume that you have already gotten a MTLDevice using MTLCreateSystemDefaultDevice() or
// MTLCopyAllDevices(), used it to create a MTLCommandQueue with MTLDevice.newCommandQueue, and
// similarly made textures with the device as needed.
void  MyBlurTextureInPlace( id <MTLTexture> __strong *inTexture, float blurRadius, id <MTLCommandQueue> q)
{
   // Create "the usual Metal objects".
   // MPS does not need a dedicated MTLCommandBuffer or MTLComputeCommandEncoder.
   // This is a trivial example. You should reuse the MTL objects you already have, if you have them.
   id <MTLDevice> device = q.device;
   id <MTLCommandBuffer> buffer = [q commandBuffer];

   // Create a MPS filter.
   MPSImageGaussianBlur *blur = [[MPSImageGaussianBlur alloc]
     initWithDevice: device];
   if( nil == blur )
       MyHandleError(kOutOfMemory);

   // Set all MPSKernel properties to taste.
   blur.sigma = blurRadius;
   // defaults are okay here for other MPSKernel properties. (clipRect, origin, edgeMode)

   // Attempt to do the work in place.  Since we provided a copyAllocator as an out-of-place
   // fallback, we don't need to check to see if it succeeded or not.
   [ blur encodeToCommandBuffer: commandBuffer
                  inPlaceTexture: inTexture         // may replace *inTexture
                   copyAllocator: myAllocator ];    // See MPSCopyAllocator definition for a sample
      myAllocator
   [ blur release];

   // the usual metal enqueue process
   [buffer waitUntilCompleted];    // slow!  Try enqueing more work on this or the next
                                   // command buffer instead of waiting every time.

   return result;
}
```

1.10 MPS Tuning Hints

MPS has been tuned for excellent performance across a diversity of devices and filter parameters. The tuning process focuses on minimizing both CPU and GPU latency for back to back calls on the same MTLCommmandBuffer. It is possible, however, to inadvertently undo this optimization effort by introducing costly operations into the pipeline around the MPS filter, leading to disappointing overall results.

Here are some elements of good practice to avoid common pitfalls:

1. Don't wait for results to complete before enqueuing more work. There can be a significant delay (up to 2.5 ms) just to get an empty MTLCommandBuffer through the pipeline to where [MTLCommandBuffer waitUntilCompleted] returns. Instead, start encoding the next command buffer(s) while you wait for the first one to complete. Enqueue them too, so they can start immediately after the previous one exits the GPU. Don't wait for the CPU kernel to notice the first command buffer is done and start taking it apart and eventually make a callback to userland before beginning work on encoding the next one. By allowing the CPU and GPU to work concurrently in this way, throughput can be enhanced by up to a factor of ten.

2. There is a large cost to allocating buffers and textures. The cost can swamp the CPU, preventing you from keeping the GPU busy. Try to preallocate and reuse MTLResource objects as much as possible. The MPSTemporaryImage may be used instead for short-lived dynamic allocations.

3. There is a cost to switching between render and compute encoders. Each time a new render encoder is used, there can be a substantial GPU mode switch cost that may undermine your throughput. To avoid the cost, try to batch compute work together. Since making a new MTLCommandBuffer forces you to make a new MTLCommandEncoder too, try to do more work with fewer MTLCommandBuffers.

4. On currently available iOS devices we find that for some image operations, particularly those involving multiple passes - for example, if you are chaining multiple MPS image filters together — performance can be improved by up a factor of two by breaking the work into tiles about 512 kB in size. Use -sourceRegionForDestinationSize: to find the MPSRegion needed for each tile.

Chapter 2

Hierarchical Index

2.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

MPSImageHistogramInfo . . . 76
MPSOffset . . . 110
MPSOrigin . . . 111
MPSRegion . . . 112
MPSScaleTransform . . . 112
MPSSize . . . 113
<NSCopying>
  MPSCNNConvolutionDescriptor . . . 25
  MPSKernel . . . 100
    MPSBinaryImageKernel . . . 17
    MPSCNNKernel . . . 32
      MPSCNNConvolution . . . 21
        MPSCNNFullyConnected . . . 29
      MPSCNNCrossChannelNormalization . . . 27
      MPSCNNLocalContrastNormalization . . . 35
      MPSCNNLogSoftMax . . . 37
      MPSCNNNeuron . . . 38
        MPSCNNNeuronAbsolute . . . 39
        MPSCNNNeuronLinear . . . 39
        MPSCNNNeuronReLU . . . 41
        MPSCNNNeuronSigmoid . . . 43
        MPSCNNNeuronTanH . . . 44
      MPSCNNPooling . . . 45
        MPSCNNPoolingAverage . . . 48
        MPSCNNPoolingMax . . . 49
      MPSCNNSoftMax . . . 50
      MPSCNNSpatialNormalization . . . 50
    MPSImageHistogram . . . 72
    MPSMatrixMultiplication . . . 108
    MPSUnaryImageKernel . . . 116
      MPSImageAreaMax . . . 57
        MPSImageAreaMin . . . 60
      MPSImageBox . . . 61
        MPSImageTent . . . 89
      MPSImageConversion . . . 62
      MPSImageConvolution . . . 64
      MPSImageDilate . . . 67

MPSImageErode . . . 69
MPSImageGaussianBlur . . . 70
MPSImageHistogramEqualization . . . 74
MPSImageHistogramSpecification . . . 77
MPSImageIntegral . . . 80
MPSImageIntegralOfSquares . . . 80
MPSImageLanczosScale . . . 81
MPSImageLaplacian . . . 82
MPSImageMedian . . . 83
MPSImagePyramid . . . 85
MPSImageGaussianPyramid . . . 72
MPSImageSobel . . . 87
MPSImageThresholdBinary . . . 89
MPSImageThresholdBinaryInverse . . . 92
MPSImageThresholdToZero . . . 94
MPSImageThresholdToZeroInverse . . . 95
MPSImageThresholdTruncate . . . 97
MPSImageTranspose . . . 99
NSObject
MPSCNNConvolutionDescriptor . . . 25
MPSImage . . . 53
MPSTemporaryImage . . . 113
MPSImageDescriptor . . . 65
MPSKernel . . . 100
MPSMatrix . . . 104
MPSMatrixDescriptor . . . 106

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

MPSBinaryImageKernel . . . 17
MPSCNNConvolution . . . 21
MPSCNNConvolutionDescriptor . . . 25
MPSCNNCrossChannelNormalization . . . 27
MPSCNNFullyConnected . . . 29
MPSCNNKernel . . . 32
MPSCNNLocalContrastNormalization . . . 35
MPSCNNLogSoftMax . . . 37
MPSCNNNeuron . . . 38
MPSCNNNeuronAbsolute . . . 39
MPSCNNNeuronLinear . . . 39
MPSCNNNeuronReLU . . . 41
MPSCNNNeuronSigmoid . . . 43
MPSCNNNeuronTanH . . . 44
MPSCNNPooling . . . 45
MPSCNNPoolingAverage . . . 48
MPSCNNPoolingMax . . . 49
MPSCNNSoftMax . . . 50
MPSCNNSpatialNormalization . . . 50
MPSImage . . . 53
MPSImageAreaMax . . . 57
MPSImageAreaMin . . . 60
MPSImageBox . . . 61
MPSImageConversion . . . 62
MPSImageConvolution . . . 64
MPSImageDescriptor . . . 65
MPSImageDilate . . . 67
MPSImageErode . . . 69
MPSImageGaussianBlur . . . 70
MPSImageGaussianPyramid . . . 72
MPSImageHistogram . . . 72
MPSImageHistogramEqualization . . . 74
MPSImageHistogramInfo
Specifies information to compute the histogram for channels of an image . . . 76
MPSImageHistogramSpecification . . . 77
MPSImageIntegral . . . 80
MPSImageIntegralOfSquares . . . 80
MPSImageLanczosScale . . . 81

MPSImageLaplacian 82
MPSImageMedian 83
MPSImagePyramid 85
MPSImageSobel 87
MPSImageTent 89
MPSImageThresholdBinary 89
MPSImageThresholdBinaryInverse 92
MPSImageThresholdToZero 94
MPSImageThresholdToZeroInverse 95
MPSImageThresholdTruncate 97
MPSImageTranspose 99
MPSKernel 100
MPSMatrix 104
MPSMatrixDescriptor 106
MPSMatrixMultiplication 108
MPSOffset 110
MPSOrigin 111
MPSRegion 112
MPSScaleTransform 112
MPSSize 113
MPSTemporaryImage 113
MPSUnaryImageKernel 116

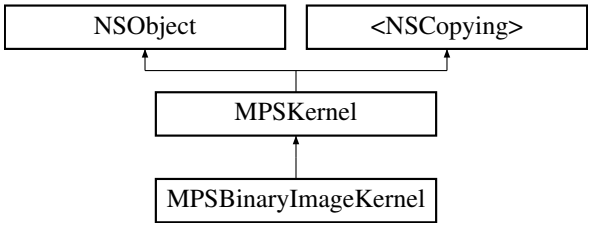
Chapter 4

Class Documentation

4.1 MPSBinaryImageKernel Class Reference

`#import <MPSImageKernel.h>`

Inheritance diagram for MPSBinaryImageKernel:



Instance Methods

- (BOOL) - encodeToCommandBuffer:primaryTexture:inPlaceSecondaryTexture:fallbackCopyAllocator:
- (BOOL) - encodeToCommandBuffer:inPlacePrimaryTexture:secondaryTexture:fallbackCopyAllocator:
- (void) - encodeToCommandBuffer:primaryTexture:secondaryTexture:destinationTexture:
- (MPSRegion) - primarySourceRegionForDestinationSize:
- (MPSRegion) - secondarySourceRegionForDestinationSize:

Properties

- MPSOffset primaryOffset
- MPSOffset secondaryOffset
- MPSImageEdgeMode primaryEdgeMode
- MPSImageEdgeMode secondaryEdgeMode
- MTLRegion clipRect

Additional Inherited Members

4.1.1 Detailed Description

This depends on Metal.framework A MPSBinaryImageKernel consumes two MTLTextures and produces one MT↩ LTexture.

4.1.2 Method Documentation

4.1.2.1 - (BOOL) encodeToCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* inPlacePrimaryTexture:(__nonnull id< MTLTexture > __strong *__nonnull) *inPlacePrimaryTexture* secondaryTexture:(nonnull id< MTLTexture >) *secondaryTexture* fallbackCopyAllocator:(nullable MPSCopyAllocator) *copyAllocator*

This method attempts to apply the MPSKernel in place on a texture.

```
In-place operation means that the same texture is used both to hold the input
image and the results. Operating in-place can be an excellent way to reduce
resource utilization, and save time and energy. While simple Metal kernels can
not operate in place because textures can not be readable and writable at the
same time, some MPSKernels can operate in place because they use
multi-pass algorithms. Whether a MPSKernel can operate in-place can
depend on current hardware, operating system revision and the parameters
and properties passed to it. You should never assume that a MPSKernel will
continue to work in place, even if you have observed it doing so before.
```

If the operation succeeds in-place, YES is returned. If the in-place operation fails and no copyAllocator is provided, then NO is returned. In neither case is the pointer held at *texture modified.

Failure during in-place operation is common. You may find it simplifies your code to provide a copyAllocator. When an in-place filter fails, your copyAllocator will be invoked to create a new texture in which to write the results, allowing the filter to proceed reliably out-of-place. The original texture will be released, replaced with a pointer to the new texture and YES will be returned. If the allocator returns an invalid texture, it is released, *texture remains unmodified and NO is returned. Please see the MPSCopyAllocator definition for a sample allocator implementation.

Note: Image filters that look at neighboring pixel values may actually consume more memory when operating in place than out of place. Many such operations are tiled internally to save intermediate texture storage, but can not tile when operating in place. The memory savings for tiling is however very short term, typically the lifetime of the MTLCommandBuffer.

Attempt to apply a MPSKernel to a texture in place.

Parameters

commandBuffer	A valid MTLCommandBuffer to receive the encoded filter
inPlace↩ PrimaryTexture	A pointer to a valid MTLTexture containing secondary image. On success, the image contents and possibly texture itself will be replaced with the result image.
secondary↩ Texture	A pointer to a valid MTLTexture containing the primary source image. It will not be overwritten.
copyAllocator	An optional block to allocate a new texture to hold the results, in case in-place operation is not possible. The allocator may use a different MTLPixelFormat or size than the original texture. You may enqueue operations on the provided MTLCommandBuffer using the provided MT↩ LComputeCommandEncoder to initialize the texture contents.

Returns

On success, YES is returned. The texture may have been replaced with a new texture if a copyAllocator was provided. On failure, NO is returned. The texture is unmodified.

4.1.2.2 - (BOOL) encodeToCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* primaryTexture:(nonnull id< MTLTexture >) *primaryTexture* inPlaceSecondaryTexture:(__nonnull id< MTLTexture > __strong *__nonnull) *inPlaceSecondaryTexture* fallbackCopyAllocator:(nullable MPSCopyAllocator) *copyAllocator*

This method attempts to apply the MPSKernel in place on a texture.

```
In-place operation means that the same texture is used both to hold the input
image and the results. Operating in-place can be an excellent way to reduce
resource utilization, and save time and energy. While simple Metal kernels can
not operate in place because textures can not be readable and writable at the
```

```
same time, some MPSKernels can operate in place because they use
multi-pass algorithms. Whether a MPSKernel can operate in-place can
depend on current hardware, operating system revision and the parameters
and properties passed to it. You should never assume that a MPSKernel will
continue to work in place, even if you have observed it doing so before.
```

If the operation succeeds in-place, YES is returned. If the in-place operation fails and no copyAllocator is provided, then NO is returned. In neither case is the pointer held at ∗texture modified.

Failure during in-place operation is common. You may find it simplifies your code to provide a copyAllocator. When an in-place filter fails, your copyAllocator will be invoked to create a new texture in which to write the results, allowing the filter to proceed reliably out-of-place. The original texture will be released, replaced with a pointer to the new texture and YES will be returned. If the allocator returns an invalid texture, it is released, ∗texture remains unmodified and NO is returned. Please see the MPSCopyAllocator definition for a sample allocator implementation.

Note: Image filters that look at neighboring pixel values may actually consume more memory when operating in place than out of place. Many such operations are tiled internally to save intermediate texture storage, but can not tile when operating in place. The memory savings for tiling is however very short term, typically the lifetime of the MTLCommandBuffer.

Attempt to apply a MPSKernel to a texture in place.

Parameters

commandBuffer	A valid MTLCommandBuffer to receive the encoded filter
primaryTexture	A pointer to a valid MTLTexture containing the primary source image. It will not be overwritten.
inPlace↩ Secondary↩ Texture	A pointer to a valid MTLTexture containing secondary image. On success, the image contents and possibly texture itself will be replaced with the result image.
copyAllocator	An optional block to allocate a new texture to hold the results, in case in-place operation is not possible. The allocator may use a different MTLPixelFormat or size than the original texture. You may enqueue operations on the provided MTLCommandBuffer using the provided MT↩ LComputeCommandEncoder to initialize the texture contents.

Returns

On success, YES is returned. The texture may have been replaced with a new texture if a copyAllocator was provided. On failure, NO is returned. The texture is unmodified.

4.1.2.3 - (void) encodeToCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* primaryTexture:(nonnull id< MTLTexture >) *primaryTexture* secondaryTexture:(nonnull id< MTLTexture >) *secondaryTexture* destinationTexture:(nonnull id< MTLTexture >) *destinationTexture*

Encode a MPSKernel into a command Buffer. The operation shall proceed out-of-place.

Parameters

commandBuffer	A valid MTLCommandBuffer to receive the encoded filter
primaryTexture	A valid MTLTexture containing the primary source image.
secondary↩ Texture	A valid MTLTexture containing the secondary source image.
destination↩ Texture	A valid MTLTexture to be overwritten by result image. destinationTexture may not alias sourceTexture.

4.1.2.4 - (MPSRegion) primarySourceRegionForDestinationSize: (MTLSize) *destinationSize*

primarySourceRegionForDestinationSize: is used to determine which region of the primaryTexture will be read by encodeToCommandBuffer:primaryTexture:secondaryTexture:destinationTexture (and in-place variants) when the filter runs. This information may be needed if the primary source image is broken into multiple textures. The size

of the full (untiled) destination image is provided. The region of the full (untiled) source image that will be read is returned. You can then piece together an appropriate texture containing that information for use in your tiled context.

The function will consult the MPSBinaryImageKernel primaryOffset and clipRect parameters, to determine the full region read by the function. Other parameters such as kernelHeight and kernelWidth will be consulted as necessary. All properties should be set to intended values prior to calling primarySourceRegionForDestinationSize:.

```
Caution: This function operates using global image coordinates, but
-encodeToCommandBuffer:... uses coordinates local to the source and
destination image textures. Consequently, the primaryOffset and clipRect
attached to this object will need to be updated using a global to
local coordinate transform before -encodeToCommandBuffer:... is
called.
```

Determine the region of the source texture that will be read for a encode operation

Parameters

destinationSize	The size of the full virtual destination image.

Returns

The area in the virtual source image that will be read.

4.1.2.5 - (MPSRegion) secondarySourceRegionForDestinationSize: (MTLSize) *destinationSize*

secondarySourceRegionForDestinationSize: is used to determine which region of the sourceTexture will be read by encodeToCommandBuffer:primaryTexture:secondaryTexture:destinationTexture (and in-place variants) when the filter runs. This information may be needed if the secondary source image is broken into multiple textures. The size of the full (untiled) destination image is provided. The region of the full (untiled) secondary source image that will be read is returned. You can then piece together an appropriate texture containing that information for use in your tiled context.

The function will consult the MPSBinaryImageKernel secondaryOffset and clipRect parameters, to determine the full region read by the function. Other parameters such as kernelHeight and kernelWidth will be consulted as necessary. All properties should be set to intended values prior to calling secondarySourceRegionForDestinationSize:.

```
Caution: This function operates using global image coordinates, but
-encodeToCommandBuffer:... uses coordinates local to the source and
destination image textures. Consequently, the secondaryOffset and clipRect
attached to this object will need to be updated using a global to
local coordinate transform before -encodeToCommandBuffer:... is
called.
```

Determine the region of the source texture that will be read for a encode operation

Parameters

destinationSize	The size of the full virtual destination image.

Returns

The area in the virtual source image that will be read.

4.1.3 Property Documentation

4.1.3.1 - clipRect [read],[write],[nonatomic],[assign]

An optional clip rectangle to use when writing data. Only the pixels in the rectangle will be overwritten. A MTL↩ Region that indicates which part of the destination to overwrite. If the clipRect does not lie completely within the destination image, the intersection between clip rectangle and destination bounds is used. Default: MPSRectNoClip (MPSKernel::MPSRectNoClip) indicating the entire image.

See Also: MPSKernel clipRect

4.1.3.2 - primaryEdgeMode [read], [write], [nonatomic], [assign]

The MPSImageEdgeMode to use when texture reads stray off the edge of the primary source image Most MPS↩ Kernel objects can read off the edge of a source image. This can happen because of a negative offset property, because the offset + clipRect.size is larger than the source image or because the filter looks at neighboring pixels, such as a Convolution or morphology filter. Default: usually MPSImageEdgeModeZero. (Some MPSKernel types default to MPSImageEdgeModeClamp, because MPSImageEdgeModeZero is either not supported or would produce unexpected results.)

See Also: MPSKernelEdgeMode

4.1.3.3 - primaryOffset [read], [write], [nonatomic], [assign]

The position of the destination clip rectangle origin relative to the primary source buffer. The offset is defined to be the position of clipRect.origin in source coordinates. Default: {0,0,0}, indicating that the top left corners of the clipRect and primary source image align.

See Also: subsubsection_mpsoffset

4.1.3.4 - secondaryEdgeMode [read], [write], [nonatomic], [assign]

The MPSImageEdgeMode to use when texture reads stray off the edge of the secondary source image Most MP↩ SKernel objects can read off the edge of a source image. This can happen because of a negative offset property, because the offset + clipRect.size is larger than the source image or because the filter looks at neighboring pixels, such as a Convolution or morphology filter. Default: usually MPSImageEdgeModeZero. (Some MPSKernel types default to MPSImageEdgeModeClamp, because MPSImageEdgeModeZero is either not supported or would produce unexpected results.)

See Also: MPSKernelEdgeMode

4.1.3.5 - secondaryOffset [read], [write], [nonatomic], [assign]

The position of the destination clip rectangle origin relative to the secondary source buffer. The offset is defined to be the position of clipRect.origin in source coordinates. Default: {0,0,0}, indicating that the top left corners of the clipRect and primary source image align.

See Also: subsubsection_mpsoffset

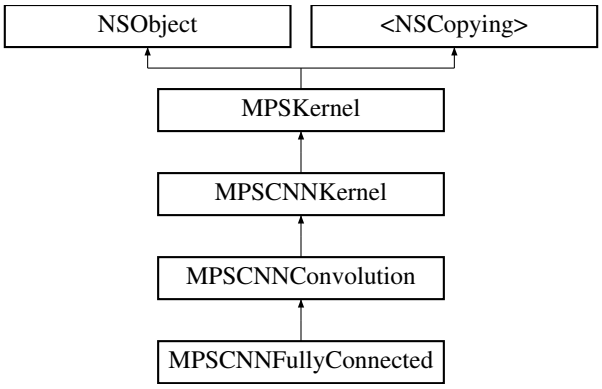
The documentation for this class was generated from the following file:

- MPSImageKernel.h

4.2 MPSCNNConvolution Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNConvolution:



Instance Methods

- (nonnull instancetype) - initWithDevice:convolutionDescriptor:kernelWeights:biasTerms:flags:
- (nonnull instancetype) - initWithDevice:

Properties

- NSUInteger kernelWidth
- NSUInteger kernelHeight
- NSUInteger inputFeatureChannels
- NSUInteger outputFeatureChannels
- NSUInteger strideInPixelsX
- NSUInteger strideInPixelsY
- NSUInteger groups
- const MPSCNNNeuron ∗__nullable neuron

Additional Inherited Members

4.2.1 Detailed Description

This depends on Metal.framework The MPSCNNConvolution specifies a convolution. The MPSCNNConvolution convolves the input image with a set of filters, each producing one feature map in the output image.

4.2.2 Method Documentation

4.2.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

Reimplemented in MPSCNNFullyConnected.

4.2.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* convolutionDescriptor:(const MPSCNNConvolutionDescriptor *__nonnull) *convolutionDescriptor* kernelWeights:(const float *__nonnull) *kernelWeights* biasTerms:(const float *__nullable) *biasTerms* flags:(MPSCNNConvolutionFlags) *NS_DESIGNATED_INITIALIZER*

Initializes a convolution kernel

Parameters

device	The MTLDevice on which this MPSCNNConvolution filter will be used
convolution↩ Descriptor	A pointer to a MPSCNNConvolutionDescriptor.
kernelWeights	A pointer to a weights array. Each entry is a float value. The number of entries is = input↩ FeatureChannels ∗ outputFeatureChannels ∗ kernelHeight ∗ kernelWidth The layout of filter weight is so that it can be reinterpreted as 4D tensor (array) weight[outputChannels][kernelHeight][kernelWidth][inputChannels / groups] Weights are converted to half float (fp16) internally for best performance.
biasTerms	A pointer to bias terms to be applied to the convolution output. Each entry is a float value. The number of entries is = numberOfOutputFeatureMaps
flags	Currently unused. Pass MPSCNNConvolutionFlagsNone

Returns

A valid MPSCNNConvolution object or nil, if failure.

Reimplemented in MPSCNNFullyConnected.

4.2.3 Property Documentation

4.2.3.1 - groups [read], [nonatomic], [assign]

Number of groups input and output channels are divided into.

4.2.3.2 - inputFeatureChannels [read], [nonatomic], [assign]

The number of feature channels per pixel in the input image.

4.2.3.3 - kernelHeight [read], [nonatomic], [assign]

The height of the filter window. Any positive non-zero value is valid, including even values. The position of the top edge of the filter window is given by offset.y - (kernelHeight>>1)

4.2.3.4 - kernelWidth [read], [nonatomic], [assign]

The width of the filter window. The position of the left edge of the filter window is given by offset.x - (kernel↩ Width>>1)

4.2.3.5 - neuron [read], [nonatomic], [assign]

MPSCNNNeuron filter to be applied as part of convolution. Can be nil in wich case no neuron activation fuction is applied.

4.2.3.6 - outputFeatureChannels [read], [nonatomic], [assign]

The number of feature channels per pixel in the output image.

4.2.3.7 - strideInPixelsX [read], [nonatomic], [assign]

The output stride (downsampling factor) in the x dimension. The default value is 1.

4.2.3.8 - strideInPixelsY [read],[nonatomic],[assign]

The output stride (downsampling factor) in the y dimension. The default value is 1.

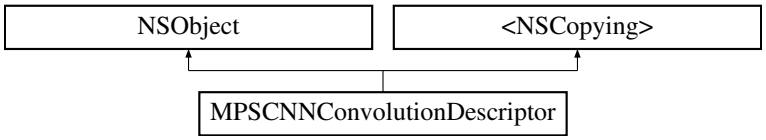
The documentation for this class was generated from the following file:

- MPSCNN.h

4.3 MPSCNNConvolutionDescriptor Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNConvolutionDescriptor:



Class Methods

- (nonnull instancetype) + cnnConvolutionDescriptorWithKernelWidth:kernelHeight:inputFeatureChannels↩ :outputFeatureChannels:neuronFilter:

Properties

- NSUInteger kernelWidth
- NSUInteger kernelHeight
- NSUInteger inputFeatureChannels
- NSUInteger outputFeatureChannels
- NSUInteger strideInPixelsX
- NSUInteger strideInPixelsY
- NSUInteger groups
- const MPSCNNNeuron ∗__nullable neuron

4.3.1 Detailed Description

This depends on Metal.framework The MPSCNNConvolutionDescriptor specifies a convolution descriptor

4.3.2 Method Documentation

4.3.2.1 + (nonnull instancetype) cnnConvolutionDescriptorWithKernelWidth: (NSUInteger) *kernelWidth* kernelHeight:(NSUInteger) *kernelHeight* inputFeatureChannels:(NSUInteger) *inputFeatureChannels* outputFeatureChannels:(NSUInteger) *outputFeatureChannels* neuronFilter:(const **MPSCNNNeuron** ∗__nullable) *neuronFilter*

Creates a convolution descriptor with a neuron filter

Parameters

kernelWidth	The width of the filter window. Must be $>$ 0. Large values will take a long time.
kernelHeight	The height of the filter window. Must be $>$ 0. Large values will take a long time.
inputFeature↩ Channels	The number of feature channels in the input image. Must be $>=$ 1.
outputFeature↩ Channels	The number of feature channels in the output image. Must be $>=$ 1.
neuronFilter	An optional neuron filter that can be applied to the output of convolution.

Returns

A valid MPSCNNConvolutionDescriptor object or nil, if failure.

4.3.3 Property Documentation

4.3.3.1 - groups [read], [write], [nonatomic], [assign]

Number of groups input and output channels are divided into. The default value is 1. Groups lets you reduce the parametrization. If groups is set to n, input is divided into n groups with inputFeatureChannels/n channels in each group. Similarly output is divided into n groups with outputFeatureChannels/n channels in each group. ith group in input is only connected to ith group in output so number of weights (parameters) needed is reduced by factor of n. Both inputFeatureChannels and outputFeatureChannels must be divisible by n and number of channels in each group must be multiple of 4.

4.3.3.2 - inputFeatureChannels [read], [write], [nonatomic], [assign]

The number of feature channels per pixel in the input image.

4.3.3.3 - kernelHeight [read], [write], [nonatomic], [assign]

The height of the filter window. The default value is 3. Any positive non-zero value is valid, including even values. The position of the top edge of the filter window is given by offset.y - (kernelHeight$>>$1)

4.3.3.4 - kernelWidth [read], [write], [nonatomic], [assign]

The width of the filter window. The default value is 3. Any positive non-zero value is valid, including even values. The position of the left edge of the filter window is given by offset.x - (kernelWidth$>>$1)

4.3.3.5 - neuron [read], [write], [nonatomic], [retain]

MPSCNNNeuron filter to be applied as part of convolution. Default is nil.

4.3.3.6 - outputFeatureChannels [read], [write], [nonatomic], [assign]

The number of feature channels per pixel in the output image.

4.3.3.7 - strideInPixelsX [read], [write], [nonatomic], [assign]

The output stride (downsampling factor) in the x dimension. The default value is 1.

4.3.3.8 - strideInPixelsY [read],[write],[nonatomic],[assign]

The output stride (downsampling factor) in the y dimension. The default value is 1.

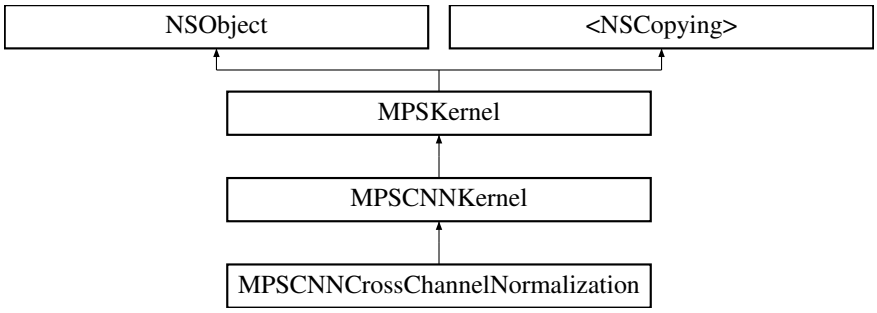
The documentation for this class was generated from the following file:

- MPSCNN.h

4.4 MPSCNNCrossChannelNormalization Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNCrossChannelNormalization:



Instance Methods

- (nonnull instancetype) - initWithDevice:kernelSize:
- (nonnull instancetype) - initWithDevice:

Properties

- float alpha
- float beta
- float delta
- NSUInteger kernelSize

Additional Inherited Members

4.4.1 Detailed Description

This depends on Metal.framework Specifies the normalization filter across feature channels. This normalization filter applies the filter to a local region across nearby feature channels, but with no spatial extent (i.e., they have shape kernelSize x 1 x 1). The normalized output is given by: Y(i,j,k) = X(i,j,k) / L(i,j,k)^beta, where the normalizing factor is: L(i,j,k) = delta + alpha/N ∗ (sum_{q in Q(k)} X(i,j,q)^2, where N is the kernel size. The window Q(k) itself is defined as: Q(k) = [max(0, k-floor(N/2)), min(D-1, k+floor((N-1)/2)], where

k is the feature channel index (running from 0 to D-1) and D is the number of feature channels, and alpha, beta and delta are paremeters. It is the end-users responsibility to ensure that the combination of the parameters delta and alpha does not result in a situation where the denominator becomes zero - in such situations the resulting pixel-value is undefined.

4.4.2 Method Documentation

4.4.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.4.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* kernelSize:(NSUInteger) *NS_DESIGNATED_INITIALIZER*

Initialize a local response normalization filter in a channel

Parameters

device	The device the filter will run on
kernelSize	The kernel filter size in each dimension.

Returns

A valid MPSCNNCrossFeatureMapNormalization object or nil, if failure.

4.4.3 Property Documentation

4.4.3.1 - alpha [read], [write], [nonatomic], [assign]

The value of alpha. Default is 1.0. Must be non-negative.

4.4.3.2 - beta [read], [write], [nonatomic], [assign]

The value of beta. Default is 5.0

4.4.3.3 - delta [read], [write], [nonatomic], [assign]

The value of delta. Default is 1.0

4.4.3.4 - kernelSize [read], [nonatomic], [assign]

The size of the square filter window. Default is 5

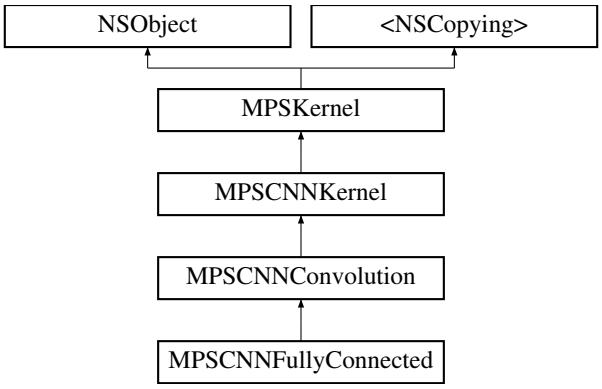
The documentation for this class was generated from the following file:

- MPSCNN.h

4.5 MPSCNNFullyConnected Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNFullyConnected:



Instance Methods

- (nonnull instancetype) - initWithDevice:convolutionDescriptor:kernelWeights:biasTerms:flags:
- (nonnull instancetype) - initWithDevice:

Additional Inherited Members

4.5.1 Detailed Description

This depends on Metal.framework The MPSCNNFullyConnected specifies a fully connected convolution layer a.k.a. Inner product layer Fully connected layer in CNN is one where every input channel is connected to every output channel. Kernel width is equal to width of source image and kernel height is equal to height of source image. Width and height of output is 1x1 Thus it takes srcW x srcH x Ni MPSCNNImage, convolves it with Weights[No][Src↩W][srcH][Ni] and produces 1 x 1 x No output. Thus following must be true kernelWidth == source.width kernelHeight == source.height clipRect.size.width == 1 clipRect.size.height == 1 One can think of fully connected layer as matrix multiplication where image is flattened into a vector of length srcW∗srcH∗Ni and weights are arragned in a matrix of dimension No x (srcW∗srcH∗Ni) to product output vector of length No. strideInPixelsX, strideInPixelsX, group must to 1. offset is not applicable and ignored. Since clipRect clamped to destination image bounds, if the destination is 1x1, one doesn't need to set clipRect.

Note that one can implement inner product using MPSCNNConvolution by setting offset = (kernelWidth/2,kernel↩Height/2) clipRect.origin = (ox,oy), clipRect.size = (1,1) strideX = strideX = group = 1 However using MPSCNN↩FullyConnected for this better for performance as it lets us choose best approach to implement it which may not be possible when its implemented using general convolution. For example, we may internally use matrix multiplication or special reduction kernels for a specific platform.

4.5.2 Method Documentation

4.5.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSCNNConvolution.

4.5.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* convolutionDescriptor:(const MPSCNNConvolutionDescriptor *__nonnull) *fullyConnectedDescriptor* kernelWeights:(const float *__nonnull) *kernelWeights* biasTerms:(const float *__nullable) *biasTerms* flags:(MPSCNNConvolutionFlags) *NS_DESIGNATED_INITIALIZER*

Initializes a fully connected kernel.

Parameters

device	The MTLDevice on which this MPSCNNFullyConnected filter will be used
fully↩ Connected↩ Descriptor	A pointer to a MPSCNNConvolutionDescriptor. strideInPixelsX, strideInPixelsY and group properties of fullyConnectedDescriptor must be set to 1 (default).
kernelWeights	A pointer to a weights array. Each entry is a float value. The number of entries is = input↩ FeatureChannels ∗ outputFeatureChannels ∗ kernelHeight ∗ kernelWidth The layout of filter weight is so that it can be reinterpreted as 4D tensor (array) weight[outputChannels][kernelHeight][kernelWidth][inputChannels / groups] Weights are converted to half float (fp16) internally for best performance.
biasTerms	A pointer to bias terms to be applied to the convolution output. Each entry is a float value. The number of entries is = numberOfOutputFeatureMaps
flags	Currently unused. Pass MPSCNNConvolutionFlagsNone

Returns

A valid MPSCNNConvolution object or nil, if failure.

Reimplemented from MPSCNNConvolution.

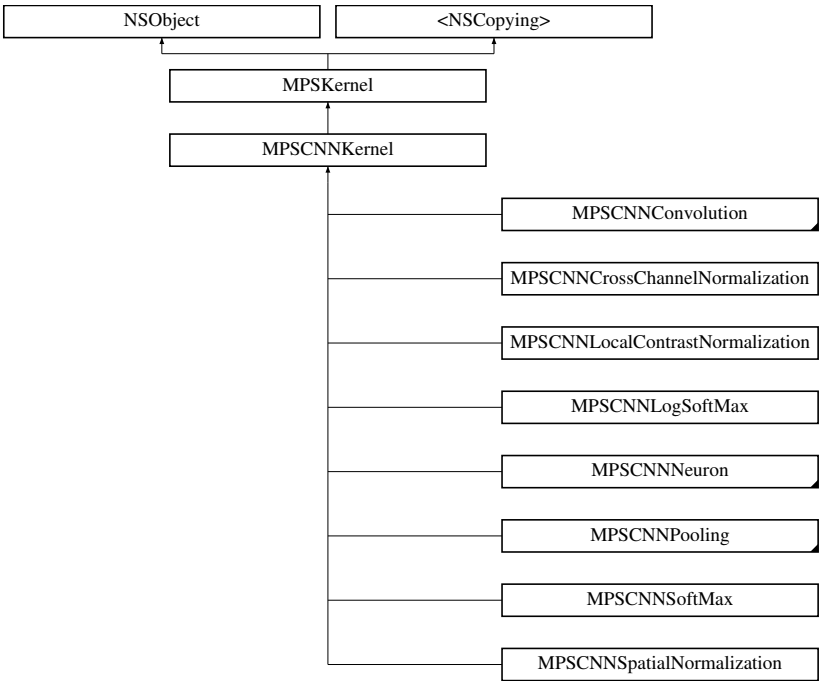
The documentation for this class was generated from the following file:

- MPSCNN.h

4.6 MPSCNNKernel Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNKernel:



Instance Methods

- (void) - encodeToCommandBuffer:sourceImage:destinationImage:
- (MPSRegion) - sourceRegionForDestinationSize:

Properties

- MPSOffset offset
- MTLRegion clipRect
- NSUInteger destinationFeatureChannelOffset
- MPSImageEdgeMode edgeMode

Additional Inherited Members

4.6.1 Detailed Description

MPSCNN.h MetalPerformanceShaders.framework

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved. Metal Performance Shaders CNN kernels

This depends on Metal.framework Describes a convolution neural network kernel. A MPSCNNKernel consumes one MPSImage and produces one MPSImage.

```
The region overwritten in the destination MPSImage is described
by the clipRect.  The top left corner of the region consumed (ignoring
adjustments for filter size -- e.g. convolution filter size) is given
by the offset. The size of the region consumed is a function of the
clipRect size and any subsampling caused by pixel strides at work,
e.g. MPSCNNPooling.strideInPixelsX/Y.  Where the offset + clipRect
would cause a {x,y} pixel address not in the image to be read, the
edgeMode is used to determine what value to read there.

The Z/depth component of the offset, clipRect.origin and clipRect.size
indexes which images to use. If the MPSImage contains only a single image
then these should be offset.z = 0, clipRect.origin.z = 0
and clipRect.size.depth = 1. If the MPSImage contains multiple images,
clipRect.size.depth refers to number of images to process. Both source
and destination MPSImages must have at least this many images. offset.z
refers to starting source image index. Thus offset.z + clipRect.size.depth must
be <= source.numberOfImages. Similarly, clipRect.origin.z refers to starting
image index in destination. So clipRect.origin.z + clipRect.size.depth must be
<= destination.numberOfImage.

destinationFeatureChannelOffset property can be used to control where the MPSKernel will
start writing in feature channel dimension. For example, if the destination image has
64 channels, and MPSKernel outputs 32 channels, by default channels 0-31 of destination
will be populated by MPSKernel. But if we want this MPSKernel to populate channel 32-63
of the destination, we can set destinationFeatureChannelOffset = 32.
A good example of this is concat (concatenation) operation in Tensor Flow. Suppose
we have a src = w x h x Ni which goes through CNNConvolution_0 which produces
output O0 = w x h x N0 and CNNConvolution_1 which produces output O1 = w x h x N1 followed
by concatenation which produces O = w x h x (N0 + N1). We can achieve this by creating
an MPSImage with dimensions O = w x h x (N0 + N1) and using this as destination of
both convolutions as follows
    CNNConvolution0: destinationFeatureChannelOffset = 0, this will output N0 channels starting at
                     channel 0 of destination thus populating [0,N0-1] channels.
    CNNConvolution1: destinationFeatureChannelOffset = N0, this will output N1 channels starting at
                     channel N0 of destination thus populating [N0,N0+N1-1] channels.
```

4.6.2 Method Documentation

4.6.2.1 - (void) encodeToCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* sourceImage:(MPSImage *__nonnull) *sourceImage* destinationImage:(encode(commandBuffer:sourceImage:destinationImage:)) *MPS_SWIFT_NAME*

Encode a MPSCNNKernel into a command Buffer. The operation shall proceed out-of-place.

Parameters

commandBuffer	A valid MTLCommandBuffer to receive the encoded filter
sourceImage	A valid MPSImage object containing the source image.
destinationImage	A valid MPSImage to be overwritten by result image. destinationImage may not alias sourceImage.

4.6.2.2 - (MPSRegion) sourceRegionForDestinationSize: (sourceRegion(destinationSize:)) *MPS_SWIFT_NAME*

sourceRegionForDestinationSize: is used to determine which region of the sourceTexture will be read by encodeToCommandBuffer:sourceImage:destinationImage (and similar) when the filter runs. This information may be needed if the source image is broken into multiple textures. The size of the full (untiled) destination image is provided. The region of the full (untiled) source image that will be read is returned. You can then piece together an appropriate texture containing that information for use in your tiled context.

The function will consult the MPSCNNKernel offset and clipRect parameters, to determine the full region read by the function. Other parameters such as sourceClipRect, kernelHeight and kernelWidth will be consulted as necessary. All properties should be set to intended values prior to calling sourceRegionForDestinationSize:.

```
Caution: This function operates using global image coordinates, but
-encodeToCommandBuffer:... uses coordinates local to the source and
destination image textures. Consequently, the offset and clipRect
attached to this object will need to be updated using a global to
local coordinate transform before -encodeToCommandBuffer:... is
called.
```

Determine the region of the source texture that will be read for a encode operation

Parameters

destinationSize	The size of the full virtual destination image.

Returns

The area in the virtual source image that will be read.

4.6.3 Property Documentation

4.6.3.1 - clipRect [read], [write], [nonatomic], [assign]

An optional clip rectangle to use when writing data. Only the pixels in the rectangle will be overwritten. A MTLRegion that indicates which part of the destination to overwrite. If the clipRect does not lie completely within the destination image, the intersection between clip rectangle and destination bounds is used. Default: MPSRectNoClip (MPSKernel::MPSRectNoClip) indicating the entire image. clipRect.origin.z is the index of starting destination image in batch processing mode. clipRect.size.depth is the number of images to process in batch processing mode.

See Also: MPSKernel clipRect

4.6.3.2 - destinationFeatureChannelOffset [read], [write], [nonatomic], [assign]

The number of channels in the destination MPSImage to skip before writing output. This is the starting offset into the destination image in the feature channel dimension at which destination data is written. This allows an application to pass a subset of all the channels in MPSImage as output of MPSKernel. E.g. Suppose MPSImage has 24 channels and a MPSKernel outputs 8 channels. If we want channels 8 to 15 of this MPSImage to be used as output, we can set destinationFeatureChannelOffset = 8. Note that this offset applies independently to each image when the MPSImage is a container for multiple images and the MPSCNNKernel is processing multiple images (clipRect.size.depth $>$ 1). The default value is 0 and any value specifed shall be a multiple of 4. If MPSKernel outputs N channels, destination image MUST have at least destinationFeatureChannelOffset + N channels. Using a destination image with insufficient number of feature channels result in an error. E.g. if the MPSCNNConvolution

outputs 32 channels, and destination has 64 channels, then it is an error to set destinationFeatureChannelOffset > 32.

4.6.3.3 - edgeMode [read], [write], [nonatomic], [assign]

The MPSImageEdgeMode to use when texture reads stray off the edge of an image Most MPSKernel objects can read off the edge of the source image. This can happen because of a negative offset property, because the offset + clipRect.size is larger than the source image or because the filter looks at neighboring pixels, such as a Convolution filter. Default: MPSImageEdgeModeZero.

See Also: MPSKernelEdgeMode Note: For MPSCNNPoolingAverage specifying edge mode MPSImageEdge↩ ModeClamp is interpreted as a "shrink-to-edge" operation, which shrinks the effective filtering window to remain within the source image borders.

4.6.3.4 - offset [read], [write], [nonatomic], [assign]

The position of the destination clip rectangle origin relative to the source buffer. The offset is defined to be the position of clipRect.origin in source coordinates. Default: {0,0,0}, indicating that the top left corners of the clipRect and source image align. offset.z is the index of starting source image in batch processing mode.

See Also: subsubsection_mpsoffset

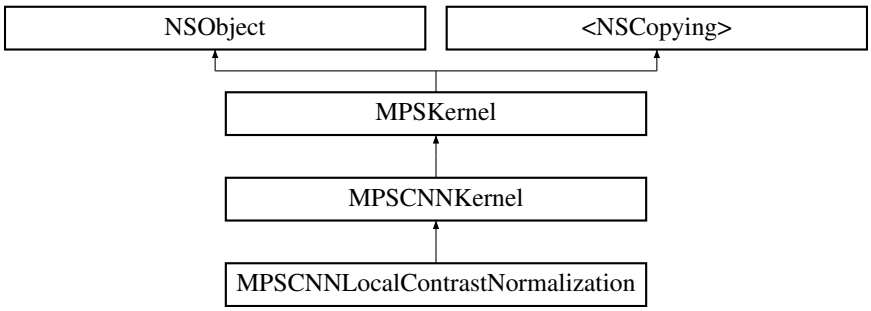
The documentation for this class was generated from the following file:

- MPSCNN.h

4.7 MPSCNNLocalContrastNormalization Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNLocalContrastNormalization:



Instance Methods

- (nonnull instancetype) - initWithDevice:kernelWidth:kernelHeight:
- (nonnull instancetype) - initWithDevice:

Properties

- float alpha
- float beta
- float delta
- float p0
- float pm

- float ps
- NSUInteger kernelWidth
- NSUInteger kernelHeight

Additional Inherited Members

4.7.1 Detailed Description

This depends on Metal.framework Specifies the local contrast normalization filter. The local contrast normalization is quite similar to spatial normalization (see MPSCNNSpatialNormalization) in that it applies the filter over local regions which extend spatially, but are in separate feature channels (i.e., they have shape 1 x kernelWidth x kernelHeight), but instead of dividing by the local "energy" of the feature, the denominator uses the local variance of the feature - effectively the mean value of the feature is subtracted from the signal. For each feature channel, the function computes the variance VAR(i,j) and mean M(i,j) of X(i,j) inside each rectangle around the spatial point (i,j).

Then the result is computed for each element of X as follows:

```
Y(i,j) = pm + ps * ( X(i,j) - p0 * M(i,j)) / (delta + alpha * VAR(i,j))^beta,
```

where kw and kh are the kernelWidth and the kernelHeight and pm, ps and p0 are parameters that can be used to offset and scale the result in various ways. For example setting pm=0, ps=1, p0=1, delta=0, alpha=1.0 and beta=0.5 scales input data so that the result has unit variance and zero mean, provided that input variance is positive. It is the end-users responsibility to ensure that the combination of the parameters delta and alpha does not result in a situation where the denominator becomes zero - in such situations the resulting pixel-value is undefined. A good way to guard against tiny variances is to regulate the expression with a small value for delta, for example delta = 1/1024 = 0.0009765625.

4.7.2 Method Documentation

4.7.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.7.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* kernelWidth:(NSUInteger) *kernelWidth* kernelHeight:(NSUInteger) *NS_DESIGNATED_INITIALIZER*

Initialize a local contrast normalization filter

Parameters

device	The device the filter will run on
kernelWidth	The width of the kernel

kernelHeight	The height of the kernel

Returns

A valid MPSCNNLocalContrastNormalization object or nil, if failure.

NOTE: For now, kernelWidth must be equal to kernelHeight

4.7.3 Property Documentation

4.7.3.1 - alpha [read], [write], [nonatomic], [assign]

The value of alpha. Default is 1.0

4.7.3.2 - beta [read], [write], [nonatomic], [assign]

The value of beta. Default is 0.5

4.7.3.3 - delta [read], [write], [nonatomic], [assign]

The value of delta. Default is 1/1024

4.7.3.4 - kernelHeight [read], [nonatomic], [assign]

The height of the filter window

4.7.3.5 - kernelWidth [read], [nonatomic], [assign]

The width of the filter window

4.7.3.6 - p0 [read], [write], [nonatomic], [assign]

The value of p0. Default is 1.0

4.7.3.7 - pm [read], [write], [nonatomic], [assign]

The value of pm. Default is 0.0

4.7.3.8 - ps [read], [write], [nonatomic], [assign]

The value of ps. Default is 1.0

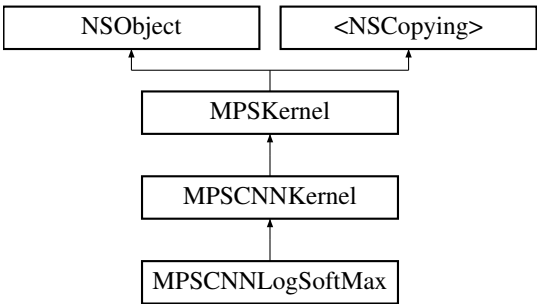
The documentation for this class was generated from the following file:

- MPSCNN.h

4.8 MPSCNNLogSoftMax Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNLogSoftMax:



Additional Inherited Members

4.8.1 Detailed Description

This depends on Metal.framework The logarithmic softmax filter can be achieved by taking the natural logarithm of the the result of the softmax filter. The results are often used to construct a loss function to be minimized when training neural networks. For each feature channel per pixel in an image in a feature map, the logarithmic softmax filter computes the following: result channel in pixel = pixel(x,y,k)) - ln{sum(exp(pixel(x,y,0)) ... exp(pixel(x,y,N-1))} where N is the number of feature channels and y = ln{x} satisfies e^y = x.

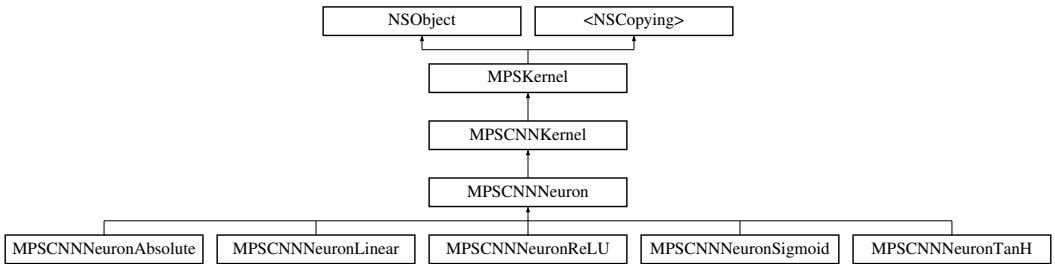
The documentation for this class was generated from the following file:

- MPSCNN.h

4.9 MPSCNNNeuron Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNNeuron:



Additional Inherited Members

4.9.1 Detailed Description

This depends on Metal.framework This filter applies a neuron activation function. You must use one of the subclasses of MPSCNNNeuron

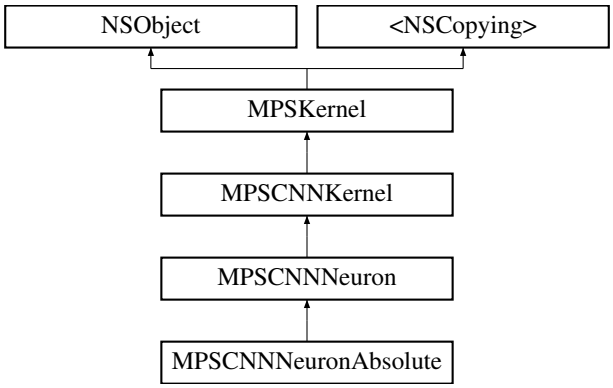
The documentation for this class was generated from the following file:

- MPSCNN.h

4.10 MPSCNNNeuronAbsolute Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNNeuronAbsolute:



Instance Methods

- (nonnull instancetype) - initWithDevice:

Additional Inherited Members

4.10.1 Detailed Description

This depends on Metal.framework Specifies the absolute neuron filter. For each pixel, applies the following function: f(x) = | x |

4.10.2 Method Documentation

4.10.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Initialize a neuron filter

Parameters

device	The device the filter will run on

Returns

A valid MPSCNNNeuronAbsolute object or nil, if failure.

Reimplemented from MPSKernel.

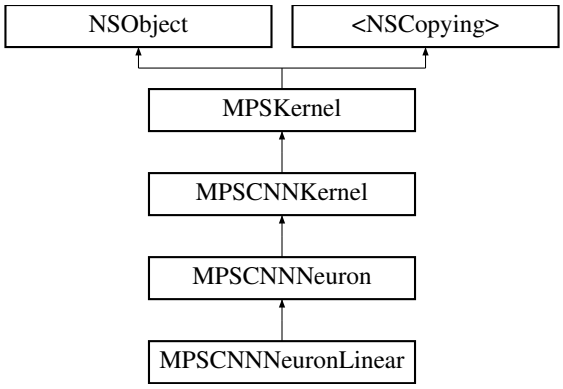
The documentation for this class was generated from the following file:

- MPSCNN.h

4.11 MPSCNNNeuronLinear Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNNeuronLinear:



Instance Methods

- (nonnull instancetype) - initWithDevice:a:b:
- (nonnull instancetype) - initWithDevice:

Properties

- float **a**
- float **b**

Additional Inherited Members

4.11.1 Detailed Description

This depends on Metal.framework Specifies the linear neuron filter. For each pixel, applies the following function: f(x) = a ∗ x + b

4.11.2 Method Documentation

4.11.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.11.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* a:(float) *a* b:(float) *NS_DESIGNATED_INITIALIZER*

Initialize the linear neuron filter

Parameters

device	The device the filter will run on
a	Filter property "a". See lass discussion.
b	Filter property "b". See class discussion.

Returns

A valid MPSCNNNeuronLinear object or nil, if failure.

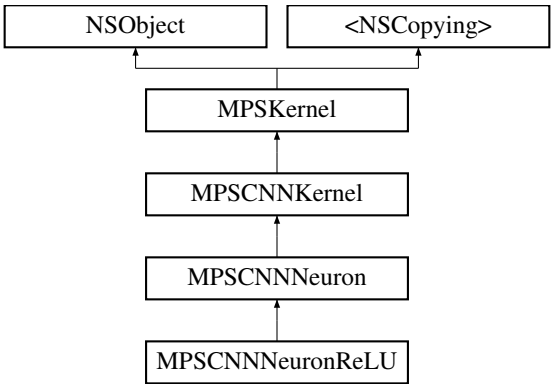
The documentation for this class was generated from the following file:

- MPSCNN.h

4.12 MPSCNNNeuronReLU Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNNeuronReLU:



Instance Methods

- (nonnull instancetype) - initWithDevice:a:
- (nonnull instancetype) - initWithDevice:

Properties

- float **a**

Additional Inherited Members

4.12.1 Detailed Description

This depends on Metal.framework Specifies the ReLU neuron filter. For each pixel, applies the following function: f(x) = x, if x >= 0 = a * x if x < 0 This is called Leaky ReLU in literature. Some literature defines classical ReLU as max(0, x). If you want this behavior, simply pass a = 0

4.12.2 Method Documentation

4.12.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.12.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* a:(float) *NS_DESIGNATED_INITIALIZER*

Initialize the ReLU neuron filter

Parameters

device	The device the filter will run on
a	Filter property "a". See class discussion.

Returns

A valid MPSCNNNeuronReLU object or nil, if failure.

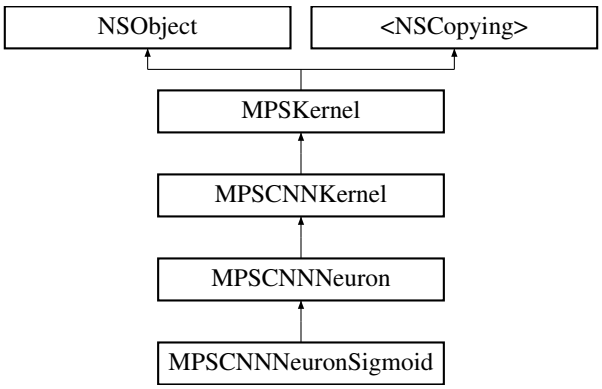
The documentation for this class was generated from the following file:

- MPSCNN.h

4.13 MPSCNNNeuronSigmoid Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNNeuronSigmoid:



Instance Methods

- (nonnull instancetype) - initWithDevice:

Additional Inherited Members

4.13.1 Detailed Description

This depends on Metal.framework Specifies the sigmoid neuron filter. For each pixel, applies the following function: f(x) = 1 / (1 + e^-x)

4.13.2 Method Documentation

4.13.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Initialize a neuron filter

Parameters

device	The device the filter will run on

Returns

A valid MPSCNNNeuronSigmoid object or nil, if failure.

Reimplemented from MPSKernel.

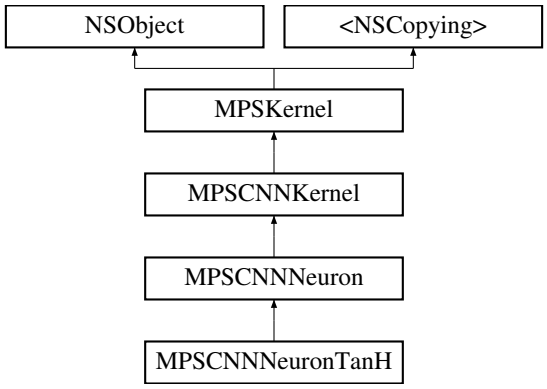
The documentation for this class was generated from the following file:

- MPSCNN.h

4.14 MPSCNNNeuronTanH Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNNeuronTanH:



Instance Methods

- (nonnull instancetype) - initWithDevice:a:b:
- (nonnull instancetype) - initWithDevice:

Properties

- float **a**
- float **b**

Additional Inherited Members

4.14.1 Detailed Description

This depends on Metal.framework Specifies the hyperbolic tangent neuron filter. For each pixel, applies the following function: f(x) = a ∗ tanh(b ∗ x)

4.14.2 Method Documentation

4.14.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.14.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* a:(float) *a* b:(float) *NS_DESIGNATED_INITIALIZER*

Initialize the hyperbolic tangent neuron filter

Parameters

device	The device the filter will run on
a	Filter property "a". See class discussion.
b	Filter property "b". See class discussion.

Returns

A valid MPSCNNNeuronTanH object or nil, if failure.

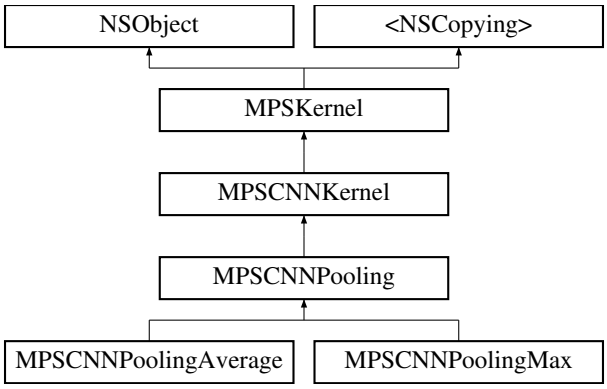
The documentation for this class was generated from the following file:

- MPSCNN.h

4.15 MPSCNNPooling Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNPooling:



Instance Methods

- (nonnull instancetype) - initWithDevice:kernelWidth:kernelHeight:
- (nonnull instancetype) - initWithDevice:kernelWidth:kernelHeight:strideInPixelsX:strideInPixelsY:
- (nonnull instancetype) - initWithDevice:

Properties

- NSUInteger kernelWidth
- NSUInteger kernelHeight
- NSUInteger strideInPixelsX
- NSUInteger strideInPixelsY

Additional Inherited Members

4.15.1 Detailed Description

This depends on Metal.framework Pooling is a form of non-linear sub-sampling. Pooling partitions the input image into a set of rectangles (overlapping or non-overlapping) and, for each such sub-region, outputs a value. The pooling operation is used in computer vision to reduce the dimensionality of intermediate representations.

4.15.2 Method Documentation

4.15.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.15.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* kernelWidth:(NSUInteger) *kernelWidth* kernelHeight:(NSUInteger) *kernelHeight*

Initialize a pooling filter

Parameters

device	The device the filter will run on
kernelWidth	The width of the kernel. Can be an odd or even value.
kernelHeight	The height of the kernel. Can be an odd or even value.

Returns

A valid MPSCNNPooling object or nil, if failure.

4.15.2.3 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* kernelWidth:(NSUInteger) *kernelWidth* kernelHeight:(NSUInteger) *kernelHeight* strideInPixelsX:(NSUInteger) *strideInPixelsX* strideInPixelsY:(NSUInteger) *NS_DESIGNATED_INITIALIZER*

Initialize a pooling filter

Parameters

device	The device the filter will run on
kernelWidth	The width of the kernel. Can be an odd or even value.
kernelHeight	The height of the kernel. Can be an odd or even value.
strideInPixelsX	The output stride (downsampling factor) in the x dimension.
strideInPixelsY	The output stride (downsampling factor) in the y dimension.

Returns

A valid MPSCNNPooling object or nil, if failure.

4.15.3 Property Documentation

4.15.3.1 - kernelHeight [read], [nonatomic], [assign]

The height of the filter window

4.15.3.2 - kernelWidth [read], [nonatomic], [assign]

The width of the filter window

4.15.3.3 - strideInPixelsX [read], [nonatomic], [assign]

The output stride (downsampling factor) in the x dimension. The default value is 1.

4.15.3.4 - strideInPixelsY [read], [nonatomic], [assign]

The output stride (downsampling factor) in the y dimension. The default value is 1.

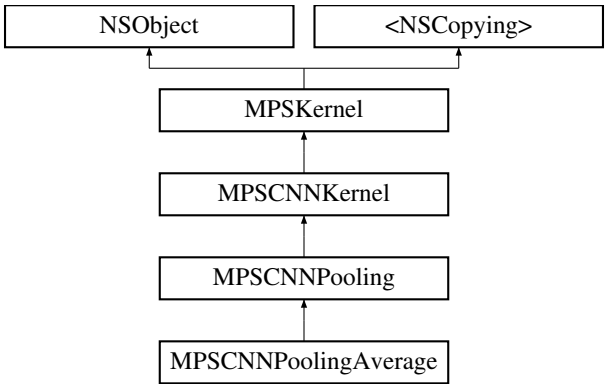
The documentation for this class was generated from the following file:

- MPSCNN.h

4.16 MPSCNNPoolingAverage Class Reference

```
#import <MPSCNN.h>
```

Inheritance diagram for MPSCNNPoolingAverage:



Additional Inherited Members

4.16.1 Detailed Description

This depends on Metal.framework Specifies the average pooling filter. For each pixel, returns the mean value of pixels in the kernelWidth x kernelHeight filter region. When edgeMode is MPSImageEdgeModeClamp the filtering window is shrunk to remain

within the source image borders. What this means is that close to image borders the filtering window

will be smaller in order to fit inside the source image and less values will be used to compute the average. In case the filtering window is entirely outside the source image border the outputted value will be zero.

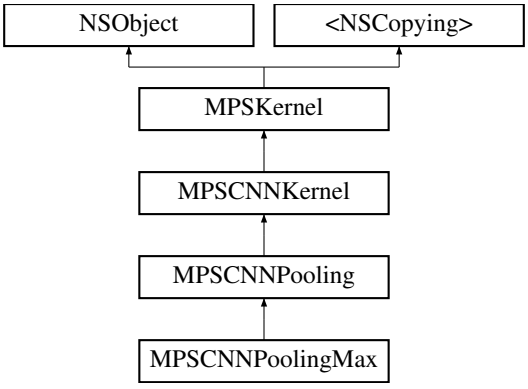
The documentation for this class was generated from the following file:

- MPSCNN.h

4.17 MPSCNNPoolingMax Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNPoolingMax:



Additional Inherited Members

4.17.1 Detailed Description

This depends on Metal.framework Specifies the max pooling filter. For each pixel, returns the maximum value of pixels in the kernelWidth x kernelHeight filter region.

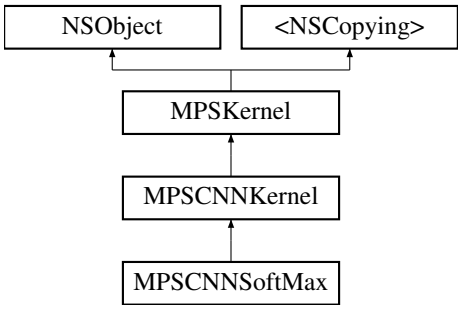
The documentation for this class was generated from the following file:

- MPSCNN.h

4.18 MPSCNNSoftMax Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNSoftMax:



Additional Inherited Members

4.18.1 Detailed Description

This depends on Metal.framework The softmax filter is a neural transfer function and is useful for classification tasks. The softmax filter is applied across feature channels and in a convolutional manner at all spatial locations. The softmax filter can be seen as the combination of an activation function (exponential) and a normalization operator. For each feature channel per pixel in an image in a feature map, the softmax filter computes the following: result channel in pixel = exp(pixel(x,y,k))/sum(exp(pixel(x,y,0)) ... exp(pixel(x,y,N-1)) where N is the number of feature channels

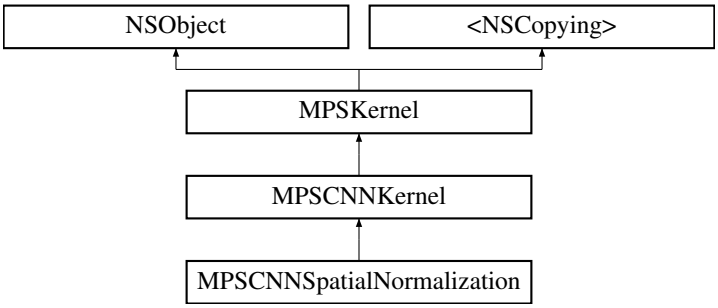
The documentation for this class was generated from the following file:

- MPSCNN.h

4.19 MPSCNNSpatialNormalization Class Reference

`#import <MPSCNN.h>`

Inheritance diagram for MPSCNNSpatialNormalization:



Instance Methods

- (nonnull instancetype) - initWithDevice:kernelWidth:kernelHeight:
- (nonnull instancetype) - initWithDevice:

Properties

- float alpha
- float beta
- float delta
- NSUInteger kernelWidth
- NSUInteger kernelHeight

Additional Inherited Members

4.19.1 Detailed Description

This depends on Metal.framework Specifies the spatial normalization filter. The spatial normalization for a feature channel applies the filter over local regions which extend spatially, but are in separate feature channels (i.e., they have shape 1 x kernelWidth x kernelHeight). For each feature channel, the function computes the sum of squares of X inside each rectangle, N2(i,j). It then divides each element of X as follows: Y(i,j) = X(i,j) / (delta + alpha/(kw∗kh) ∗ N2(i,j))^beta, where kw and kh are the kernelWidth and the kernelHeight. It is the end-users responsibility to ensure that the combination of the parameters delta and alpha does not result in a situation where the denominator becomes zero - in such situations the resulting pixel-value is undefined.

4.19.2 Method Documentation

4.19.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.19.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* kernelWidth:(NSUInteger) *kernelWidth* kernelHeight:(NSUInteger) *NS_DESIGNATED_INITIALIZER*

Initialize a spatial normalization filter

Parameters

device	The device the filter will run on
kernelWidth	The width of the kernel
kernelHeight	The height of the kernel

Returns

A valid MPSCNNSpatialNormalization object or nil, if failure.

NOTE: For now, kernelWidth must be equal to kernelHeight

4.19.3 Property Documentation

4.19.3.1 - alpha [read], [write], [nonatomic], [assign]

The value of alpha. Default is 1.0. Must be non-negative.

4.19.3.2 - beta [read], [write], [nonatomic], [assign]

The value of beta. Default is 5.0

4.19.3.3 - delta [read], [write], [nonatomic], [assign]

The value of delta. Default is 1.0

4.19.3.4 - kernelHeight [read], [nonatomic], [assign]

The height of the filter window

4.19.3.5 - kernelWidth [read], [nonatomic], [assign]

The width of the filter window

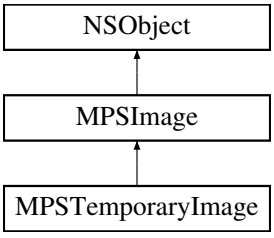
The documentation for this class was generated from the following file:

- MPSCNN.h

4.20 MPSImage Class Reference

`#import <MPSImage.h>`

Inheritance diagram for MPSImage:



Instance Methods

- (nonnull instancetype) - initWithDevice:imageDescriptor:
- (nonnull instancetype) - initWithTexture:featureChannels:
- (nonnull instancetype) - **NS_UNAVAILABLE**
- (MPSPurgeableState) - setPurgeableState:

Properties

- id< MTLDevice > device
- NSUInteger width
- NSUInteger height
- NSUInteger featureChannels
- NSUInteger numberOfImages
- MTLTextureType textureType
- MTLPixelFormat pixelFormat
- NSUInteger precision
- MTLTextureUsage usage
- size_t pixelSize
- id< MTLTexture > texture
- NSString * label

4.20.1 Detailed Description

This depends on Metal.framework A MPSImage object describes a MTLTexture that may have more than 4 channels. Some image types, such as those found in convolutional neural networks (CNN) image differs from a standard texture in that it may have more than 4 channels per image. While the channels could hold RGBA data, they will more commonly hold a number of structural permutations upon a RGBA image as the neural network progresses. It is not uncommon for each pixel to have 32 or 64 channels in it.

Since a standard MTLTexture may have no more than 4 channels, the additional channels are stored in slices of 2d texture array (i.e. texture type is MTLTextureType2DArray) such that 4 consecutive channels are stored in each slice of this array. If the number of feature channels is N, number of array slices needed is (N+3)/4. E.g. a CNN image with width 3 and height 2 with 9 channels will be stored as

```
slice 2         R???   R???   R???
                R???   R???   R???

slice 1      RGBA   RGBA   RGBA
             RGBA   RGBA   RGBA         (ASCII art /diagonal offset/ intended to show a Z dimension)

slice 0   RGBA   RGBA  RGBA
          RGBA   RGBA  RGBA
```

Thus width and height of underlying 2d texture array is same as width and height of MPSImage and array length is equal to (featureChannels + 3) / 4. Channels marked with ? are just for padding and should not contain NaNs or Infs.

MPSImage can be container of multiple CNN images for batch processing. In order to create MPSImage that contains N images, create MPSImageDescriptror with numberOfImages set to N.

Length of 2d texture array (i.e. number of slices) will be equal to ((featureChannels +3)/4)∗numberOfImage where consecutive (featureChannels+3)/4 slices of this array represent one image.

Although MPSImage can contain numberOfImages > 1, the actual number of images among these processed by MPSCNNKernel is controlled by z-dimension of clipRect. MPSCNNKernel processes n=clipRect.size.depth images from this collection. Starting index of image to process from source MPSImage is given by offset.z. Starting index of image in destination MPSImage where this processed image is written to is given by clipRect.origin.↩ z. Thus MPSCNNKernel takes n=clipRect.size.depth image from source at indices [offset.z, offset.z+n], processed each independently and stores the result in destination at indices [clipRect.origin.z, clipRect.origin.z+n] respectively.

Thus offset.z+n should be <= [src numberOfImage] and clipRect.origin.z+n should be <= [dest numberOfImages] and offset.z must be >= 0. Example: Suppose MPSCNNConvolution takes an input image with 16 channels and outputs an image with 32 channels. This number of slices needed in source 2d texture array is 2 and number of slices nneded in destination 2d array is 4. Suppose source batch size is 5 and desination batch size is 4. Thus number of source slices will be 2*5=10 and number of destination slices will be 4*4=16. If you want to process image 2 and 3 of source and store the result at index 1 and 2 in destination, you can achieve this by setting offset.z=2, clipRect.origin.z=1 and clipRect.size.depth=2. MPSCNNConvoluiton will take, in this case, slice 4 and 5 of source and produce slice 4 to 7 of destination. Similarly slice 6 and 7 will be used to produce slice 8 to 11 of destination.

All MPSCNNKernels process images in the batch independently. That is, calling a MPSCNNKernel on an batch is formally the same as calling it on each image in the batch sequentially. Computational and GPU work submission overhead will be amortized over more work if batch processing is used. This is especially important for better performance on small images.

If the number of feature channel is <= 4 and numberOfImages=1 i.e. only one slice is needed by represent MPSImage, underlying metal texture type is choosen to be MTLTextureType2D rather than MTLTextureType2DArray as explained above.

There are also MPSTemporaryImages, intended for use for very short-lived image data that is produced and consumed immediately in the same MTLCommandBuffer. They are a useful way to minimize CPU-side texture allocation costs and greatly reduce the amount of memory used by your image pipeline.

Creation of the underlying texture may in some cases occur lazily. You should in general avoid calling MPSImage.texture except when unavoidable to avoid materializing memory for longer than necessary. When possible, use the other MPSImage properties to get information about the MPSImage instead.

4.20.2 Method Documentation

4.20.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* imageDescriptor:(const MPSImageDescriptor *__nonnull) *imageDescriptor*

Initialize an empty image object

Parameters

device	The device that the image will be used. May not be NULL.
imageDescriptor	The MPSImageDescriptor. May not be NULL.

Returns

A valid MPSImage object or nil, if failure. Storage to store data needed is allocated lazily on first use of MPSImage or when application calls MPSImage.texture

Reimplemented in MPSTemporaryImage.

4.20.2.2 - (nonnull instancetype) initWithTexture: (nonnull id< MTLTexture >) *texture* featureChannels:(NSUInteger) *featureChannels*

Initialize an MPSImage object using Metal texture. Metal texture has been created by user for specific number of feature channels and number of images.

Parameters

texture	The MTLTexture allocated by the user to be used as backing for MPSImage.
featureChannels	Number of feature channels this texture contains.

Returns

A valid MPSImage object or nil, if failure. Application can let MPS framework allocate texture with properties specified in imageDescriptor using initWithDevice:MPSImageDescriptor API above. However in memory intensive application, you can save memory (and allocation/deallocation time) by using MPSTemporaryImage where MPS framework aggressively reuse memory underlying textures on same command buffer. See MPSTemporaryImage class for details below. However, in certain cases, application developer may want more control on allocation, placement, reusing/recycling of memory backing textures used in application using Metal Heaps API. In this case, application can create MPSImage from pre-allocated texture using initWithTexture:featureChannels.

MTLTextureType of texture can be MTLTextureType2D ONLY if featureChannels <= 4 in which case MPSImage.numberOfImages is set to 1. Else it should be MTLTextureType2DArray with arrayLength == numberOfImage * ((featureChannels + 3)/4). MPSImage.numberOfImages is set to texture.arrayLength / ((featureChannels + 3)/4).

For MTLTextures containing typical image data which application may obtain from MetalKit or other libraries such as that drawn from a JPEG or PNG, featureChannels should be set to number of valid color channel e.g. for RGB data, even thought MTLPixelFormat will be MTLPixelFormatRGBA, featureChannels should be set to 3.

Reimplemented in MPSTemporaryImage.

4.20.2.3 - (MPSPurgeableState) setPurgeableState: (MPSPurgeableState) *state*

setPurgeableState Set (or query) the purgeability state of a MPSImage Usage is per [MTLResource setPurgeableState:], except that the MTLTexture might be MPSPurgeableStateAllocationDeferred, which means there is no texture to mark volatile / nonvolatile. Attempts to set purgeability on MTLTextures that have not been allocated will be ignored.

4.20.3 Property Documentation

4.20.3.1 - device [read], [nonatomic], [retain]

The device on which the MPSImage will be used

4.20.3.2 - featureChannels [read], [nonatomic], [assign]

The number of feature channels per pixel.

4.20.3.3 - height [read], [nonatomic], [assign]

The formal height of the image in pixels.

4.20.3.4 - label [read], [write], [atomic], [copy]

A string to help identify this object.

4.20.3.5 - numberOfImages [read], [nonatomic], [assign]

numberOfImages for batch processing

4.20.3.6 - pixelFormat [read], [nonatomic], [assign]

The MTLPixelFormat of the underlying texture

4.20.3.7 - pixelSize [read], [nonatomic], [assign]

Number of bytes from the first byte of one pixel to the first byte of the next pixel in storage order. (Includes padding.)

4.20.3.8 - precision [read], [nonatomic], [assign]

The number of bits of numeric precision available for each feature channel. This is precision, not size. That is, float is 24 bits, not 32. half precision floating-point is 11 bits, not 16. SNorm formats have one less bit of precision for the sign bit, etc. For formats like MTLPixelFormatB5G6R5Unorm it is the precision of the most precise channel, in this case 6. When this information is unavailable, typically compressed formats, 0 will be returned.

4.20.3.9 - texture [read], [nonatomic], [assign]

The associated MTLTexture object. This is a 2D texture if numberOfImages is 1 and number of feature channels <= 4. It is a 2D texture array otherwise. To avoid the high cost of premature allocation of the underlying texture, avoid calling this property except when strictly necessary. [MPSCNNKernel encode...] calls typically cause their arguments to become allocated. Likewise, MPSImages initialized with -initWithTexture: featureChannels: have already been allocated.

4.20.3.10 - textureType [read], [nonatomic], [assign]

The type of the underlying texture, typically MTLTextureType2D or MTLTextureType2DArray

4.20.3.11 - usage [read], [nonatomic], [assign]

Description of texture usage.

4.20.3.12 - width [read], [nonatomic], [assign]

The formal width of the image in pixels.

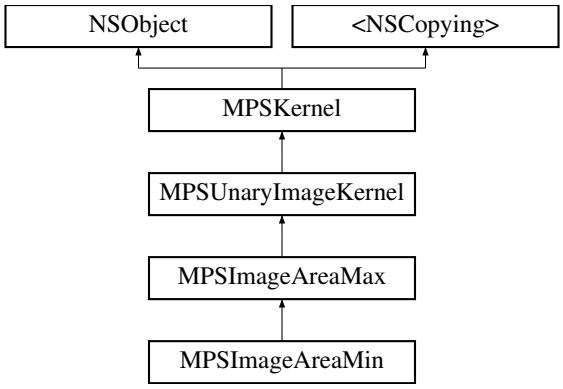
The documentation for this class was generated from the following file:

- MPSImage.h

4.21 MPSImageAreaMax Class Reference

`#import <MPSImageMorphology.h>`

Inheritance diagram for MPSImageAreaMax:



Instance Methods

- (nonnull instancetype) - initWithDevice:kernelWidth:kernelHeight:
- (nonnull instancetype) - initWithDevice:

Properties

- NSUInteger kernelHeight
- NSUInteger kernelWidth

Additional Inherited Members

4.21.1 Detailed Description

MPSImageMorphology.h MetalPerformanceShaders

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved. MetalPerformanceShaders morphological operators

The MPSImageAreaMax kernel finds the maximum pixel value in a rectangular region centered around each pixel in the source image. If there are multiple channels in the source image, each channel is processed independently. The edgeMode property is assumed to always be MPSImageEdgeModeClamp for this filter.

4.21.2 Method Documentation

4.21.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.21.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* kernelWidth:(NSUInteger) *kernelWidth* kernelHeight:(NSUInteger) *NS_DESIGNATED_INITIALIZER*

Set the kernel height and width

Parameters

device	The device the filter will run on
kernelWidth	The width of the kernel. Must be an odd number.
kernelHeight	The height of the kernel. Must be an odd number.

4.21.3 Property Documentation

4.21.3.1 - kernelHeight `[read],[nonatomic],[assign]`

The height of the filter window. Must be an odd number.

4.21.3.2 - kernelWidth `[read],[nonatomic],[assign]`

The width of the filter window. Must be an odd number.

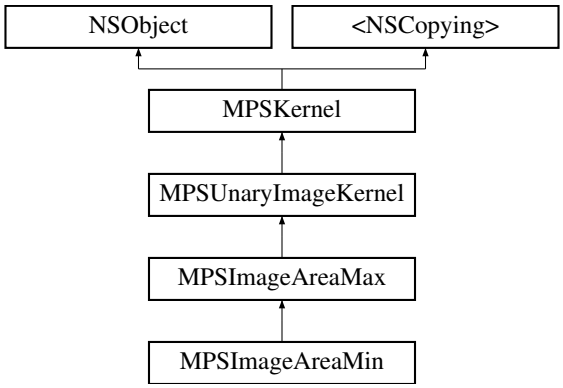
The documentation for this class was generated from the following file:

- MPSImageMorphology.h

4.22 MPSImageAreaMin Class Reference

`#import <MPSImageMorphology.h>`

Inheritance diagram for MPSImageAreaMin:



Additional Inherited Members

4.22.1 Detailed Description

The MPSImageAreaMin finds the minimum pixel value in a rectangular region centered around each pixel in the source image. If there are multiple channels in the source image, each channel is processed independently. It has the same methods as MPSImageAreaMax The edgeMode property is assumed to always be MPSImageEdge↩ ModeClamp for this filter.

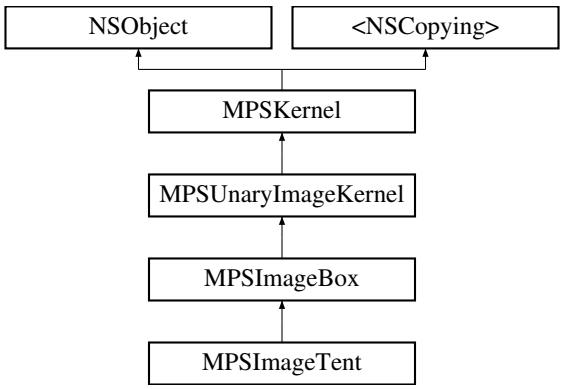
The documentation for this class was generated from the following file:

- MPSImageMorphology.h

4.23 MPSImageBox Class Reference

`#import <MPSImageConvolution.h>`

Inheritance diagram for MPSImageBox:



Instance Methods

- (nonnull instancetype) - initWithDevice:kernelWidth:kernelHeight:
- (nonnull instancetype) - initWithDevice:

Properties

- NSUInteger kernelHeight
- NSUInteger kernelWidth

Additional Inherited Members

4.23.1 Detailed Description

The MPSImageBox convolves an image with given filter of odd width and height. The kernel elements all have equal weight, achieving a blur effect. (Each result is the unweighted average of the surrounding pixels.) This allows for much faster algorithms, espcially for larger blur radii. The box height and width must be odd numbers. The box blur is a separable filter. The implementation is aware of this and will act accordingly to give best performance for multi-dimensional blurs.

4.23.2 Method Documentation

4.23.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.23.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* kernelWidth:(NSUInteger) *kernelWidth* kernelHeight:(NSUInteger) *NS_DESIGNATED_INITIALIZER*

Initialize a filter for a particular kernel size and device

Parameters

device	The device the filter will run on
kernelWidth	the width of the kernel. Must be an odd number.
kernelHeight	the height of the kernel. Must be an odd number.

Returns

A valid object or nil, if failure.

4.23.3 Property Documentation

4.23.3.1 - kernelHeight `[read], [nonatomic], [assign]`

The height of the filter window.

4.23.3.2 - kernelWidth `[read], [nonatomic], [assign]`

The width of the filter window.

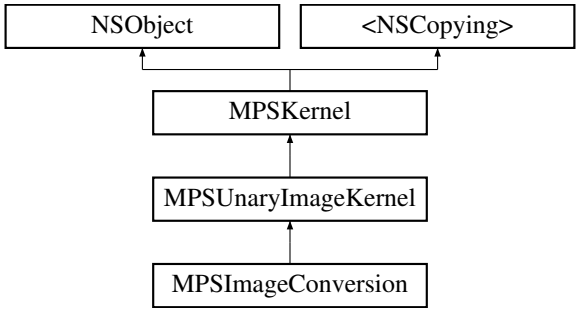
The documentation for this class was generated from the following file:

- MPSImageConvolution.h

4.24 MPSImageConversion Class Reference

`#import <MPSImageConversion.h>`

Inheritance diagram for MPSImageConversion:



Instance Methods

- (nonnull instancetype) - initWithDevice:srcAlpha:destAlpha:backgroundColor:conversionInfo:

Properties

- MPSAlphaType sourceAlpha
- MPSAlphaType destinationAlpha

Additional Inherited Members

4.24.1 Detailed Description

MPSConversions.h MetalPerformanceShaders.framework

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved. MetalPerformanceShaders conversion filters MPS_CLAS↩S_AVAILABLE_STARTING

The MPSImageConversion filter performs a conversion from source to destination

4.24.2 Method Documentation

4.24.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* srcAlpha:(MPSAlphaType) *srcAlpha* destAlpha:(MPSAlphaType) *destAlpha* backgroundColor:(nullable CGFloat *) *backgroundColor* conversionInfo:(nullable CGColorConversionInfoRef) *conversionInfo*

Create a converter that can convert texture colorspace, alpha and texture format Create a converter that can convert texture colorspace, alpha and MTLPixelFormat. Optimized cases exist for NULL color space converter and no alpha conversion.

Parameters

device	The device the filter will run on
srcAlpha	The alpha encoding for the source texture
destAlpha	The alpha encoding for the destination texture
backgroundColor	An array of CGFloats giving the background color to use when flattening an image. The color is in the source colorspace. The length of the array is the number of color channels in the src colorspace. If NULL, use {0}.
conversionInfo	The colorspace conversion to use. May be NULL, indicating no color space conversions need to be done.

Returns

An initialized MPSImageConversion object.

4.24.3 Property Documentation

4.24.3.1 - destinationAlpha [read], [nonatomic], [assign]

Premultiplication description for the destinationAlpha texture Colorspace conversion operations produce non-premultiplied data. Use this property to tag cases where premultiplied results are required. If MPSPixelAlpha↩_AlphaIsOne is used, the alpha channel will be set to 1. Default: MPSPixelAlpha_AlphaIsOne

4.24.3.2 - sourceAlpha [read], [nonatomic], [assign]

Premultiplication description for the source texture Most colorspace conversion operations can not work directly on premultiplied data. Use this property to tag premultiplied data so that the source texture can be unpremultiplied prior to application of these transforms. Default: MPSPixelAlpha_AlphaIsOne

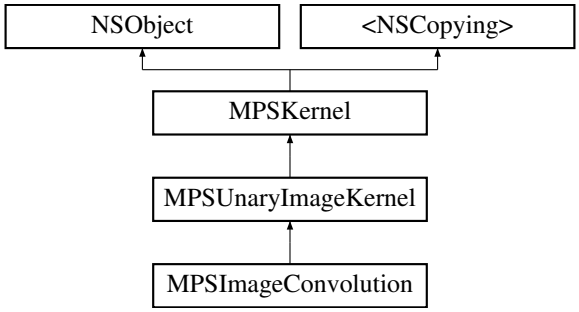
The documentation for this class was generated from the following file:

- MPSImageConversion.h

4.25 MPSImageConvolution Class Reference

`#import <MPSImageConvolution.h>`

Inheritance diagram for MPSImageConvolution:



Instance Methods

- (nonnull instancetype) - initWithDevice:kernelWidth:kernelHeight:weights:

Properties

- NSUInteger kernelHeight
- NSUInteger kernelWidth
- float bias

Additional Inherited Members

4.25.1 Detailed Description

MPSImageConvolution.h MetalPerformanceShaders

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved.

MetalPerformanceShaders Convolution Filters

The MPSImageConvolution convolves an image with given filter of odd width and height. The center of the kernel aligns with the MPSImageConvolution.offset. That is, the position of the top left corner of the area covered by the kernel is given by MPSImageConvolution.offset - {kernel_width>>1, kernel_height>>1, 0}

Optimized cases include 3x3,5x5,7x7,9x9,11x11, 1xN and Nx1. If a convolution kernel does not fall into one of these cases but is a rank-1 matrix (a.k.a. separable) then it will fall on an optimzied separable path. Other convolutions will execute with full MxN complexity.

If there are multiple channels in the source image, each channel is processed independently.

Separable convolution filters may perform better when done in two passes. A convolution filter is separable if the ratio of filter values between all rows is constant over the whole row. For example, this edge detection filter:

```
-1      0       1
-2      0       2
-1      0       1
```

can be separated into the product of two vectors:

```
1
2       x     [-1   0    1]
1
```

and consequently can be done as two, one-dimensional convolution passes back to back on the same image. In this way, the number of multiplies (ignoring the fact that we could skip zeros here) is reduced from 3∗3=9 to 3+3 = 6. There are similar savings for addition. For large filters, the savings can be profound.

4.25.2 Method Documentation

4.25.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* kernelWidth:(NSUInteger) *kernelWidth* kernelHeight:(NSUInteger) *kernelHeight* weights:(const float ∗__nonnull) *NS_DESIGNATED_INITIALIZER*

Initialize a convolution filter

Parameters

device	The device the filter will run on
kernelWidth	the width of the kernel
kernelHeight	the height of the kernel
kernelWeights	A pointer to an array of kernelWidth ∗ kernelHeight values to be used as the kernel. These are in row major order.

Returns

A valid MPSImageConvolution object or nil, if failure.

4.25.3 Property Documentation

4.25.3.1 - bias [read], [write], [nonatomic], [assign]

The bias is a value to be added to convolved pixel before it is converted back to the storage format. It can be used to convert negative values into a representable range for a unsigned MTLPixelFormat. For example, many edge detection filters produce results in the range [-k,k]. By scaling the filter weights by 0.5/k and adding 0.5, the results will be in range [0,1] suitable for use with unorm formats. It can be used in combination with renormalization of the filter weights to do video ranging as part of the convolution effect. It can also just be used to increase the brightness of the image.

Default value is 0.0f.

4.25.3.2 - kernelHeight [read], [nonatomic], [assign]

The height of the filter window. Must be an odd number.

4.25.3.3 - kernelWidth [read], [nonatomic], [assign]

The width of the filter window. Must be an odd number.

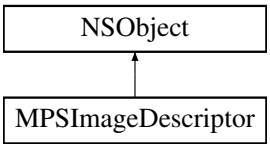
The documentation for this class was generated from the following file:

- MPSImageConvolution.h

4.26 MPSImageDescriptor Class Reference

```
#import <MPSImage.h>
```

Inheritance diagram for MPSImageDescriptor:



Class Methods

- (__nonnull instancetype) + imageDescriptorWithChannelFormat:width:height:featureChannels:
- (__nonnull instancetype) + imageDescriptorWithChannelFormat:width:height:featureChannels:numberOf↩Images:usage:

Properties

- NSUInteger width
- NSUInteger height
- NSUInteger featureChannels
- NSUInteger numberOfImages
- MTLPixelFormat pixelFormat
- MPSImageFeatureChannelFormat channelFormat
- MTLCPUCacheMode cpuCacheMode
- MTLStorageMode storageMode
- MTLTextureUsage usage

4.26.1 Detailed Description

MPSImage.h MetalPerformanceShaders.framework

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved. A MPSImage is a MTLTexture abstraction that allows for more than 4 channels, and for temporary images.

This depends on Metal.framework A MPSImageDescriptor object describes a attributes of MPSImage and is used to create one (see MPSImage discussion below)

4.26.2 Method Documentation

4.26.2.1 + (__nonnull instancetype) imageDescriptorWithChannelFormat: (MPSImageFeatureChannelFormat) *channelFormat* width:(NSUInteger) *width* height:(NSUInteger) *height* featureChannels:(NSUInteger) *featureChannels*

Create a MPSImageDescriptor for a single read/write cnn image.

4.26.2.2 + (__nonnull instancetype) imageDescriptorWithChannelFormat: (MPSImageFeatureChannelFormat) *channelFormat* width:(NSUInteger) *width* height:(NSUInteger) *height* featureChannels:(NSUInteger) *featureChannels* numberOfImages:(NSUInteger) *numberOfImages* usage:(MTLTextureUsage) *usage*

Create a MPSImageDescriptor for a read/write cnn image with option to set usage and batch size (numberOf↩Images).

4.26.3 Property Documentation

4.26.3.1 - channelFormat [read], [write], [nonatomic], [assign]

The storage format to use for each channel in the image.

4.26.3.2 - cpuCacheMode [read], [write], [nonatomic], [assign]

Options to specify CPU cache mode of texture resource. Default = MTLCPUCacheModeDefaultCache

4.26.3.3 - featureChannels [read], [write], [nonatomic], [assign]

The number of feature channels per pixel. Default = 1.

4.26.3.4 - height [read], [write], [nonatomic], [assign]

The height of the CNN image. The formal height of the CNN image in pixels. Default = 1.

4.26.3.5 - numberOfImages [read], [write], [nonatomic], [assign]

The number of images for batch processing. Default = 1.

4.26.3.6 - pixelFormat [read], [nonatomic], [assign]

The MTLPixelFormat expected for the underlying texture.

4.26.3.7 - storageMode [read], [write], [nonatomic], [assign]

To specify storage mode of texture resource. Default = MTLStorageModeShared on iOS MTLStorageModeManaged on Mac OSX MTLStorageModeShared not supported on Mac OSX. See Metal headers for synchronization requirements when using StorageModeManaged

4.26.3.8 - usage [read], [write], [nonatomic], [assign]

Description of texture usage. Default = MTLTextureUsageShaderRead/Write

4.26.3.9 - width [read], [write], [nonatomic], [assign]

The width of the CNN image. The formal width of the CNN image in pixels. Default = 1.

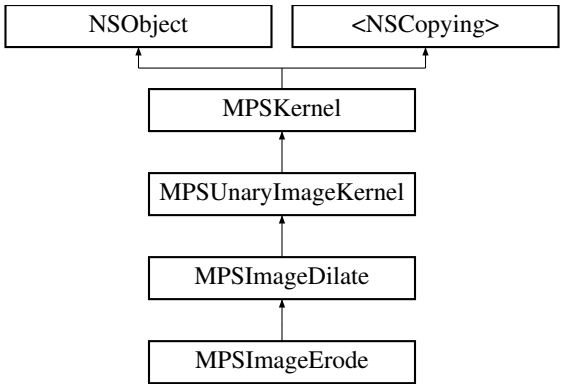
The documentation for this class was generated from the following file:

- MPSImage.h

4.27 MPSImageDilate Class Reference

`#import <MPSImageMorphology.h>`

Inheritance diagram for MPSImageDilate:



Instance Methods

- (nonnull instancetype) - initWithDevice:kernelWidth:kernelHeight:values:
- (nonnull instancetype) - initWithDevice:

Properties

- NSUInteger kernelHeight
- NSUInteger kernelWidth

Additional Inherited Members

4.27.1 Detailed Description

The MPSImageDilate finds the maximum pixel value in a rectangular region centered around each pixel in the source image. It is like the MPSImageAreaMax, except that the intensity at each position is calculated relative to a different value before determining which is the maximum pixel value, allowing for shaped, non-rectangular morphological probes.

```
for each pixel in the filter window:
    value =  pixel[filterY][filterX] - filter[filterY*filter_width+filterX]
    if( value > bestValue ){
         result = value
         bestValue = value;
    }
```

A filter that contains all zeros and is identical to a MPSImageAreaMax filter. The center filter element is assumed to be 0 to avoid causing a general darkening of the image.

The edgeMode property is assumed to always be MPSImageEdgeModeClamp for this filter.

4.27.2 Method Documentation

4.27.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.27.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* kernelWidth:(NSUInteger) *kernelWidth* kernelHeight:(NSUInteger) *kernelHeight* values:(const float *__nonnull) *NS_DESIGNATED_INITIALIZER*

Init a object with kernel height, width and weight values. Each dilate shape probe defines a 3D surface of values. These are arranged in order left to right, then top to bottom in a 1D array. (values[kernelWidth*y+x] = probe[y][x]) Values should be generally be in the range [0,1] with the center pixel tending towards 0 and edges towards 1. However, any numerical value is allowed. Calculations are subject to the usual floating-point rounding error.

Parameters

device	The device the filter will run on
kernelWidth	The width of the kernel. Must be an odd number.
kernelHeight	The height of the kernel. Must be an odd number.
values	The set of values to use as the dilate probe. The values are copied into the filter. To avoid image ligthening or darkening, the center value should be 0.0f.

4.27.3 Property Documentation

4.27.3.1 - kernelHeight [read],[nonatomic],[assign]

The height of the filter window. Must be an odd number.

4.27.3.2 - kernelWidth [read],[nonatomic],[assign]

The width of the filter window. Must be an odd number.

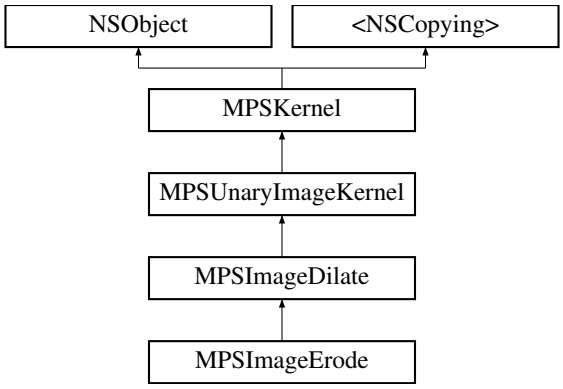
The documentation for this class was generated from the following file:

- MPSImageMorphology.h

4.28 MPSImageErode Class Reference

`#import <MPSImageMorphology.h>`

Inheritance diagram for MPSImageErode:



Additional Inherited Members

4.28.1 Detailed Description

The MPSImageErode filter finds the minimum pixel value in a rectangular region centered around each pixel in the source image. It is like the MPSImageAreaMin, except that the intensity at each position is calculated relative to a different value before determining which is the maximum pixel value, allowing for shaped, non-rectangular morphological probes.

```
for each pixel in the filter window:
    value =  pixel[filterY][filterX] + filter[filterY*filter_width+filterX]
    if( value < bestValue ){
         result = value
         bestValue = value;
    }
```

A filter that contains all zeros is identical to a MPSImageAreaMin filter. The center filter element is assumed to be 0, to avoid causing a general lightening of the image.

The definition of the filter for MPSImageErode is different from vImage. (MPSErode_filter_value = 1.0f-vImage↩ Erode_filter_value.) This allows MPSImageDilate and MPSImageErode to use the same filter, making open and close operators easier to write. The edgeMode property is assumed to always be MPSImageEdgeModeClamp for this filter.

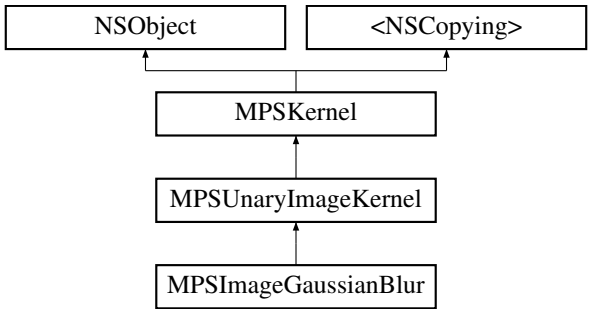
The documentation for this class was generated from the following file:

- MPSImageMorphology.h

4.29 MPSImageGaussianBlur Class Reference

`#import <MPSImageConvolution.h>`

Inheritance diagram for MPSImageGaussianBlur:



Instance Methods

- (nonnull instancetype) - initWithDevice:sigma:
- (nonnull instancetype) - initWithDevice:

Properties

- float sigma

Additional Inherited Members

4.29.1 Detailed Description

The MPSImageGaussianBlur convolves an image with gaussian of given sigma in both x and y direction.

```
The MPSImageGaussianBlur utilizes a very fast algorith that typically runs at approximately
1/2 of copy speeds. Notably, it is faster than either the tent or box blur except perhaps
for very large filter windows. Mathematically, it is an approximate gaussian. Some
non-gaussian behavior may be detectable with advanced analytical methods such as FFT.
If a analytically clean gaussian filter is required, please use the MPSImageConvolution
filter instead with an appropriate set of weights. The MPSImageGaussianBlur is intended
to be suitable for all common image processing needs demanding ~10 bits of precision or
less.
```

4.29.2 Method Documentation

4.29.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.29.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* sigma:(float) *NS_DESIGNATED_INITIALIZER*

Initialize a gaussian blur filter for a particular sigma and device

Parameters

device	The device the filter will run on
sigma	The standard deviation of gaussian blur filter. Gaussian weight, centered at 0, at integer grid i is given as w(i) = 1/sqrt(2∗pi∗sigma) ∗ exp(-i^2/2∗sigma^2) If we take cut off at 1% of w(0) (max weight) beyond which weights are considered 0, we have ceil (sqrt(-log(0.01)∗2)∗sigma) ∼ ceil(3.7∗sigma) as rough estimate of filter width

Returns

A valid object or nil, if failure.

4.29.3 Property Documentation

4.29.3.1 - sigma [read],[nonatomic],[assign]

Read-only sigma value with which filter was created

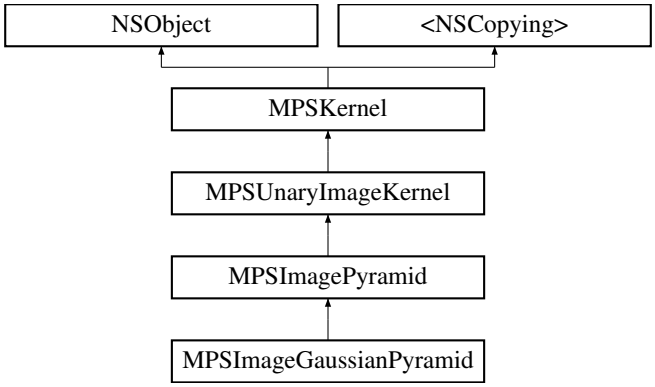
The documentation for this class was generated from the following file:

- MPSImageConvolution.h

4.30 MPSImageGaussianPyramid Class Reference

`#import <MPSImageConvolution.h>`

Inheritance diagram for MPSImageGaussianPyramid:



Additional Inherited Members

4.30.1 Detailed Description

The Gaussian image pyramid is constructed as follows: First the zeroth level mipmap of the input image is filtered with the specified convolution kernel. The default the convolution filter kernel is

```
k = w w^T, where w = [ 1/16,  1/4,  3/8,  1/4,  1/16 ]^T,
```

but the user may also tweak this kernel with a centerWeight parameter: 'a':

```
k = w w^T, where w = [ (1/4 - a/2),  1/4,  a,  1/4,  (1/4 - a/2) ]^T
```

or the user can provide a completely custom kernel. After this the image is downsampled by removing all odd rows and columns, which defines the next level in the Gaussian image pyramid. This procedure is continued until every mipmap level present in the image texture are filled with the pyramid levels.

In case of the Gaussian pyramid the user must run the operation in-place using: inPlaceTexture:fallbackCopy↩ Allocator:, where the fallback allocator is ignored.

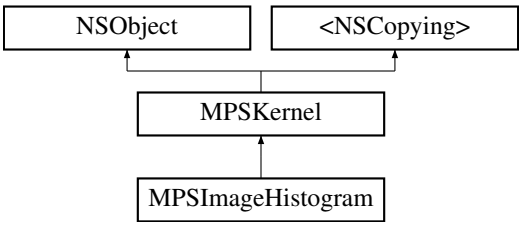
The documentation for this class was generated from the following file:

- MPSImageConvolution.h

4.31 MPSImageHistogram Class Reference

`#import <MPSImageHistogram.h>`

Inheritance diagram for MPSImageHistogram:



Instance Methods

- (nonnull instancetype) - initWithDevice:histogramInfo:
- (void) - encodeToCommandBuffer:sourceTexture:histogram:histogramOffset:
- (size_t) - histogramSizeForSourceFormat:

Properties

- MTLRegion clipRectSource
- BOOL zeroHistogram
- MPSImageHistogramInfo histogramInfo

Additional Inherited Members

4.31.1 Detailed Description

The MPSImageHistogram computes the histogram of an image.

4.31.2 Method Documentation

4.31.2.1 - (void) encodeToCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* sourceTexture:(nonnull id< MTLTexture >) *source* histogram:(nonnull id< MTLBuffer >) *histogram* histogramOffset:(NSUInteger) *histogramOffset*

Encode the filter to a command buffer using a MTLComputeCommandEncoder. The filter will not begin to execute until after the command buffer has been enqueued and committed.

Parameters

commandBuffer	A valid MTLCommandBuffer.
source	A valid MTLTexture containing the source image for the filter
histogram	A valid MTLBuffer to receive the histogram results.
histogramOffset	Byte offset into histogram buffer at which to write the histogram results. Must be a multiple of 32 bytes. The histogram results / channel are stored together. The number of channels for which histogram results are stored is determined by the number of channels in the image. If histogramInfo.histogramForAlpha is false and the source image is RGBA then only histogram results for RGB channels are stored.

4.31.2.2 - (size_t) histogramSizeForSourceFormat: (MTLPixelFormat) *sourceFormat*

The amount of space in the output MTLBuffer the histogram will take up. This convenience function calculates the minimum amount of space needed in the output histogram for the results. The MTLBuffer should be at least this length, longer if histogramOffset is non-zero.

Parameters

sourceFormat	The MTLPixelFormat of the source image. This is the source parameter of -encodeTo↩ CommandBuffer: sourceTexture:histogram:histogramOffset

Returns

The number of bytes needed to store the result histograms.

4.31.2.3 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* histogramInfo:(const MPSImageHistogramInfo *__nonnull) *NS_DESIGNATED_INITIALIZER*

Specifies information to compute the histogram for channels of an image.

Parameters

device	The device the filter will run on
histogramInfo	Pointer to the MPSHistogramInfo struct

Returns

A valid MPSImageHistogram object or nil, if failure.

4.31.3 Property Documentation

4.31.3.1 - clipRectSource [read], [write], [nonatomic], [assign]

The source rectangle to use when reading data. A MTLRegion that indicates which part of the source to read. If the clipRectSource does not lie completely within the source image, the intersection of the image bounds and clip↩ RectSource will be used. The clipRectSource replaces the MPSUnaryImageKernel offset parameter for this filter. The latter is ignored. Default: MPSRectNoClip, use the entire source texture.

4.31.3.2 - histogramInfo [read], [nonatomic], [assign]

Return a structure describing the histogram content Returns a MPSImageHistogramInfo structure describing the format of the histogram.

4.31.3.3 - zeroHistogram [read], [write], [nonatomic], [assign]

Zero-initalize the histogram results Indicates that the memory region in which the histogram results are to be written in the histogram buffer are to be zero-initialized or not. Default: YES.

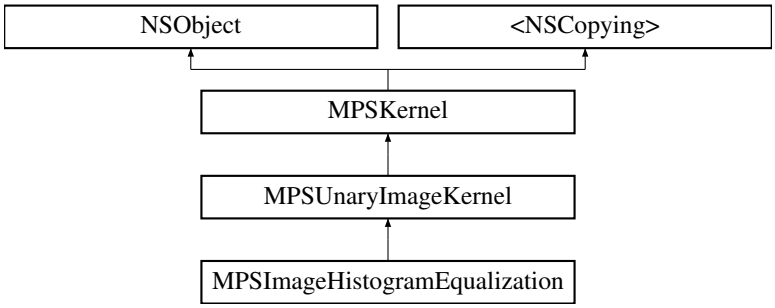
The documentation for this class was generated from the following file:

- MPSImageHistogram.h

4.32 MPSImageHistogramEqualization Class Reference

`#import <MPSImageHistogram.h>`

Inheritance diagram for MPSImageHistogramEqualization:



Instance Methods

- (nonnull instancetype) - initWithDevice:histogramInfo:
- (void) - encodeTransformToCommandBuffer:sourceTexture:histogram:histogramOffset:

Properties

- MPSImageHistogramInfo histogramInfo

Additional Inherited Members

4.32.1 Detailed Description

The MPSImageHistogramEqualization performs equalizes the histogram of an image. The process is divided into three steps.

1. Call -initWithDevice:histogramInfo: This creates a MPSImageHistogramEqualization object. It is done when the method returns.
2. Call -encodeTransform:sourceTexture:histogram:histogramOffset: This creates a privately held image transform (i.e. a cumulative distribution function of the histogram) which will be used to equalize the distribution of the histogram of the source image. This process runs on a MTLCommandBuffer when it is committed to a MTLCommandQueue. It must complete before the next step can be run. It may be performed on the same MTLCommandBuffer. The histogram argument specifies the histogram buffer which contains the histogram values for sourceTexture. The sourceTexture argument is used by encodeTransform to determine the number of channels and therefore which histogram data in histogram buffer to use. The histogram for sourceTexture must have been computed either on the CPU or using the MPSImageHistogram kernel
3. Call -encodeToCommandBuffer:sourceTexture:destinationTexture: to read data from sourceTexture, apply the equalization transform to it and write to destination texture. This step is also done on the GPU on a MTL↩ CommandQueue.

You can reuse the same equalization transform on other images to perform the same transform on those images. (Since their distribution is probably different, they will probably not be equalized by it.) This filter usually will not be able to work in place.

4.32.2 Method Documentation

4.32.2.1 - (void) encodeTransformToCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* sourceTexture:(nonnull id< MTLTexture >) *source* histogram:(nonnull id< MTLBuffer >) *histogram* histogramOffset:(NSUInteger) *histogramOffset*

Encode the transform function to a command buffer using a MTLComputeCommandEncoder. The transform function computes the equalization lookup table. The transform function will not begin to execute until after the command

buffer has been enqueued and committed. This step will need to be repeated with the new MPSKernel if -copy↩ WithZone:device or -copyWithZone: is called. The transform is stored as internal state to the object. You still need to call -encodeToCommandBuffer:sourceTexture:destinationTexture: afterward to apply the transform to produce a result texture.

Parameters

commandBuffer	A valid MTLCommandBuffer.
source	A valid MTLTexture containing the source image for the filter.
histogram	A valid MTLBuffer containing the histogram results for an image. This filter will use these histogram results to generate the cumulative histogram for equalizing the image. The histogram results / channel are stored together. The number of channels for which histogram results are stored is determined by the number of channels in the image. If histogramInfo.↩ histogramForAlpha is false and the source image is RGBA then only histogram results for RGB channels are stored.
histogramOffset	A byte offset into the histogram MTLBuffer where the histogram starts. Must conform to alignment requirements for [MTLComputeCommandEncoder setBuffer:offset:atIndex:] offset parameter.

4.32.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* histogramInfo:(const MPSImageHistogramInfo *__nonnull) *NS_DESIGNATED_INITIALIZER*

Specifies information about the histogram for the channels of an image.

Parameters

device	The device the filter will run on
histogramInfo	Pointer to the MPSHistogramInfo struct

Returns

A valid MPSImageHistogramEqualization object or nil, if failure.

4.32.3 Property Documentation

4.32.3.1 - histogramInfo [read],[nonatomic],[assign]

Return a structure describing the histogram content Returns a MPSImageHistogramInfo structure describing the format of the histogram.

The documentation for this class was generated from the following file:

- MPSImageHistogram.h

4.33 MPSImageHistogramInfo Struct Reference

Specifies information to compute the histogram for channels of an image.

`#include <MPSImageHistogram.h>`

Public Attributes

- NSUInteger numberOfHistogramEntries
- BOOL histogramForAlpha
- vector_float4 minPixelValue
- vector_float4 maxPixelValue

4.33.1 Detailed Description

Specifies information to compute the histogram for channels of an image.

MPSImageHistogram.h MetalPerformanceShaders.framework

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved. MetalPerformanceShaders histogram filters

4.33.2 Member Data Documentation

4.33.2.1 BOOL MPSImageHistogramInfo::histogramForAlpha

Specifies whether the histogram for the alpha channel should be computed or not.

4.33.2.2 vector_float4 MPSImageHistogramInfo::maxPixelValue

Specifies the maximum pixel value. Any pixel value greater than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the first histogram entry. This maximum value is applied to each of the four channels separately.

4.33.2.3 vector_float4 MPSImageHistogramInfo::minPixelValue

Specifies the minimum pixel value. Any pixel value less than this will be clipped to this value (for the purposes of histogram calculation), and assigned to the first histogram entry. This minimum value is applied to each of the four channels separately.

4.33.2.4 NSUInteger MPSImageHistogramInfo::numberOfHistogramEntries

Specifies the number of histogram entries, or "bins" for each channel. For example, if you want 256 histogram bins then numberOfHistogramEntries must be set to 256. The value stored in each histogram bin is a 32-bit unsigned integer. The size of the histogram buffer in which these bins will be stored should be >= numberOfHistogramEntries * sizeof(uint32_t) * number of channels in the image. numberOfHistogramEntries must be a power of 2 and is a minimum of 256 bins.

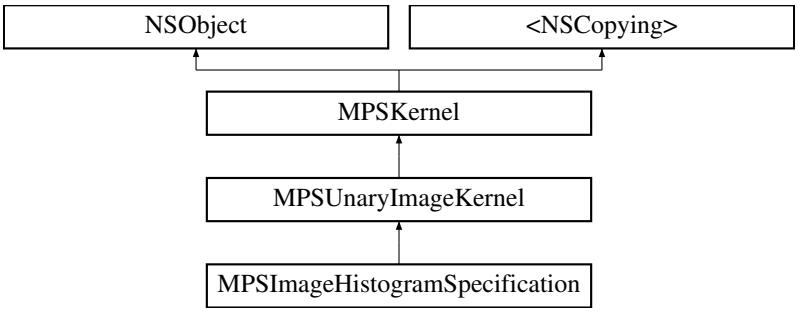
The documentation for this struct was generated from the following file:

- MPSImageHistogram.h

4.34 MPSImageHistogramSpecification Class Reference

`#import <MPSImageHistogram.h>`

Inheritance diagram for MPSImageHistogramSpecification:



Instance Methods

- (nonnull instancetype) - initWithDevice:histogramInfo:
- (void) - encodeTransformToCommandBuffer:sourceTexture:sourceHistogram:sourceHistogramOffset:desiredHistogram:desiredHistogramOffset:

Properties

- MPSImageHistogramInfo histogramInfo

Additional Inherited Members

4.34.1 Detailed Description

The MPSImageHistogramSpecification performs a histogram specification operation on an image. It is a generalized version of histogram equalization operation. The histogram specificaiton filter converts the image so that its histogram matches the desired histogram.

4.34.2 Method Documentation

4.34.2.1 - (void) encodeTransformToCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* sourceTexture:(nonnull id< MTLTexture >) *source* sourceHistogram:(nonnull id< MTLBuffer >) *sourceHistogram* sourceHistogramOffset:(NSUInteger) *sourceHistogramOffset* desiredHistogram:(nonnull id< MTLBuffer >) *desiredHistogram* desiredHistogramOffset:(NSUInteger) *desiredHistogramOffset*

Encode the transform function to a command buffer using a MTLComputeCommandEncoder. The transform function computes the specification lookup table. The transform function will not begin to execute until after the command buffer has been enqueued and committed. This step will need to be repeated with the new MPSKernel if -copyWithZone:device or -copyWithZone: is called.

Parameters

commandBuffer	A valid MTLCommandBuffer.
source	A valid MTLTexture containing the source image for the filter.
sourceHistogram	A valid MTLBuffer containing the histogram results for the source image. This filter will use these histogram results to generate the cumulative histogram for equalizing the image. The histogram results / channel are stored together. The number of channels for which histogram results are stored is determined by the number of channels in the image. If histogramInfo.histogramForAlpha is false and the source image is RGBA then only histogram results for RGB channels are stored.

sourceHistogramOffset	A byte offset into the sourceHistogram MTLBuffer where the histogram starts. Must conform to alignment requirements for [MTLComputeCommandEncoder setBuffer:offset:atIndex:] offset parameter.
desiredHistogram	A valid MTLBuffer containing the desired histogram results for the source image. The histogram results / channel are stored together. The number of channels for which histogram results are stored is determined by the number of channels in the image. If histogramInfo.histogramForAlpha is false and the source image is RGBA then only histogram results for RGB channels are stored.
desiredHistogramOffset	A byte offset into the desiredHistogram MTLBuffer where the histogram starts. Must conform to alignment requirements for [MTLComputeCommandEncoder setBuffer:offset:atIndex:] offset parameter.

4.34.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* histogramInfo:(const MPSImageHistogramInfo *__nonnull) *NS_DESIGNATED_INITIALIZER*

Specifies information about the histogram for the channels of an image. The MPSImageHistogramSpecification applies a transfor to convert the histogram to a specified histogram. The process is divided into three steps:

1. Call -initWithDevice:histogramInfo: This creates a MPSImageHistogramSpecification object. It is done when the method returns.

2. Call -encodeTransform:sourceTexture:sourceHistogram:sourceHistogramOffset:desiredHistogram: desiredHistogramOffset: This creates a privately held image transform which will convert the the distribution of the source histogram to the desired histogram. This process runs on a MTLCommandBuffer when it is committed to a MTLCommandQueue. It must complete before the next step can be run. It may be performed on the same MTLCommandBuffer. The sourceTexture argument is used by encodeTransform to determine the number of channels and therefore which histogram data in sourceHistogram buffer to use. The sourceHistogram and desiredHistogram must have been computed either on the CPU or using the MPSImageHistogram kernel

3. Call -encodeToCommandBuffer:sourceTexture:destinationTexture: to read data from sourceTexture, apply the transform to it and write to destination texture. This step is also done on the GPU on a MTLCommandQueue.

You can reuse the same specification transform on other images to perform the same transform on those images. (Since their starting distribution is probably different, they will probably not arrive at the same distribution as the desired histogram.) This filter usually will not be able to work in place.

Parameters

device	The device the filter will run on
histogramInfo	Pointer to the MPSHistogramInfo struct

Returns

A valid MPSImageHistogramSpecification object or nil, if failure.

4.34.3 Property Documentation

4.34.3.1 - histogramInfo [read], [nonatomic], [assign]

Return a structure describing the histogram content Returns a MPSImageHistogramInfo structure describing the format of the histogram.

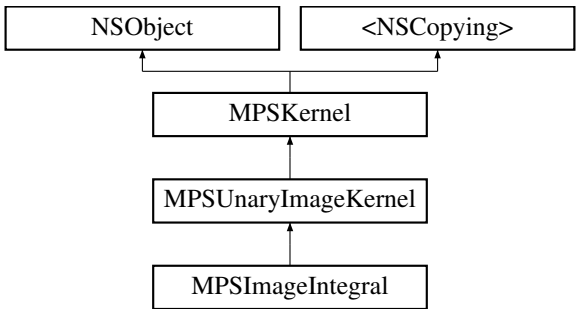
The documentation for this class was generated from the following file:

- MPSImageHistogram.h

4.35 MPSImageIntegral Class Reference

`#import <MPSImageIntegral.h>`

Inheritance diagram for MPSImageIntegral:



Additional Inherited Members

4.35.1 Detailed Description

MPSImageIntegral.h MetalPerformanceShaders.framework

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved. MetalPerformanceShaders integral filters

The MPSImageIntegral calculates the sum of pixels over a specified region in the image. The value at each position is the sum of all pixels in a source image rectangle, sumRect:

```
sumRect.origin = MPSUnaryImageKernel.offset
sumRect.size = dest_position - MPSUnaryImageKernel.clipRect.origin
```

If the channels in the source image are normalized, half-float or floating values, the destination image is recommended to be a 32-bit floating-point image. If the channels in the source image are integer values, it is recommended that an appropriate 32-bit integer image destination format is used.

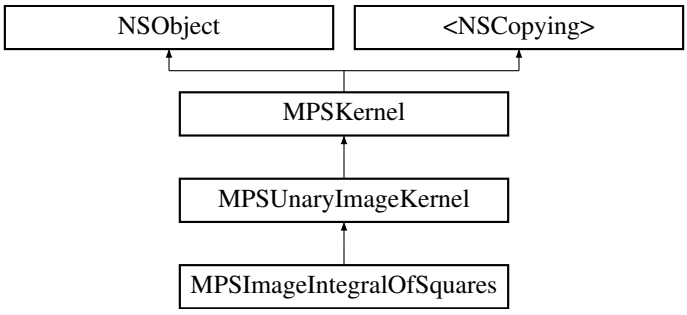
The documentation for this class was generated from the following file:

- MPSImageIntegral.h

4.36 MPSImageIntegralOfSquares Class Reference

`#import <MPSImageIntegral.h>`

Inheritance diagram for MPSImageIntegralOfSquares:



Additional Inherited Members

4.36.1 Detailed Description

The MPSImageIntegralOfSquares calculates the sum of squared pixels over a specified region in the image. The value at each position is the sum of all squared pixels in a source image rectangle, sumRect:

```
sumRect.origin = MPSUnaryImageKernel.offset
sumRect.size = dest_position - MPSUnaryImageKernel.clipRect.origin
```

If the channels in the source image are normalized, half-float or floating values, the destination image is recommended to be a 32-bit floating-point image. If the channels in the source image are integer values, it is recommended that an appropriate 32-bit integer image destination format is used.

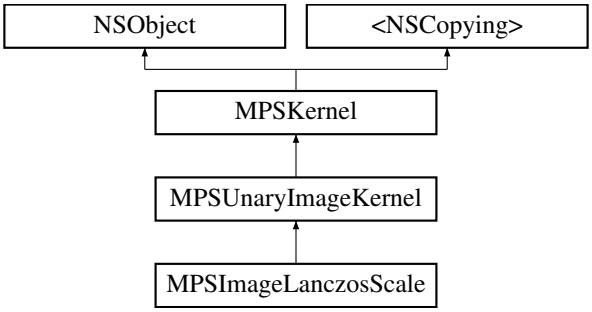
The documentation for this class was generated from the following file:

- MPSImageIntegral.h

4.37 MPSImageLanczosScale Class Reference

```
#import <MPSImageResampling.h>
```

Inheritance diagram for MPSImageLanczosScale:



Properties

- const MPSScaleTransform * scaleTransform

Additional Inherited Members

4.37.1 Detailed Description

MPSImageResampling.h MetalPerformanceShaders

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved. Resampling filters for MetalPerformanceShaders

Resize an image and / or change its aspect ratio The MPSImageLanczosScale filter can be used to resample an existing image using a different sampling frequency in each dimension. This can be used to enlarge or reduce the size of an image, or change the aspect ratio of an image. The filter uses a Lanczos resampling algorithm which typically produces better quality for photographs, but is slower than linear sampling using the GPU texture units. Lanczos downsampling does not require a low pass filter to be applied before it is used. Because the resampling function has negative lobes, Lanczos can result in ringing near sharp edges, making it less suitable for vector art.

4.37.2 Property Documentation

4.37.2.1 - scaleTransform [read],[write],[nonatomic],[assign]

An optional transform that describes how to scale and translate the source image If the scaleTransform is NU↩ LL, then the MPSImageLanczosScale filter will rescale the image so that the source image fits exactly into the destination texture. If the transform is not NULL, then the transform is used for determining how to map the source image to the destination. Default: NULL

When the scaleTransform is set to non-NULL, the values pointed to by the new scaleTransform are copied to object storage, and the pointer is updated to point to internal storage. Do not attempt to free it. You may free your copy of the MPSScaleTransform as soon as the property set operation is complete.

When calculating a scaleTransform, use the limits of the bounding box for the intended source region of interest and the destination clipRect. Adjustments for pixel center coordinates are handled internally to the function. For example, the scale transform to convert the entire source image to the entire destination image size (clipRect = MPSRectNoClip) would be:

```
scaleTransform.scaleX = (double) dest.width / source.width;
scaleTransform.scaleY = (double) dest.height / source.height;
scaleTransform.translateX = scaleTransform.translateY = 0.0;
```

The translation parameters allow you to adjust the region of the source image used to create the destination image. They are in destination coordinates. To place the top left corner of the destination clipRect to represent the position {x,y} in source coordinates, we solve for the translation based on the standard scale matrix operation for each axis:

```
dest_position = source_position * scale + translation;
translation = dest_position - source_position * scale;
```

For the top left corner of the clipRect, the dest_position is considered to be {0,0}. This gives us a translation of:

```
scaleTransform.translateX = -source_origin.x * scaleTransform.scaleX;
scaleTransform.translateY = -source_origin.y * scaleTransform.scaleY;
```

One would typically use non-zero translations to do tiling, or provide a resized view into a internal segment of an image.

Changing the Lanczos scale factor may trigger recalculation of signficant state internal to the object when the filter is encoded to the command buffer. The scale factor is scaleTransform->scaleX,Y, or the ratio of source and destination image sizes if scaleTransform is NULL. Reuse a MPSImageLancosScale object for frequently used scalings to avoid redundantly recreating expensive resampling state.

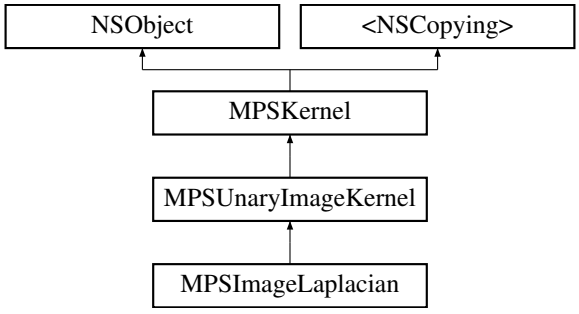
The documentation for this class was generated from the following file:

- MPSImageResampling.h

4.38 MPSImageLaplacian Class Reference

```
#import <MPSImageConvolution.h>
```

Inheritance diagram for MPSImageLaplacian:



Properties

- float bias

Additional Inherited Members

4.38.1 Detailed Description

The MPSImageLaplacian is an optimized variant of the MPSImageConvolution filter provided primarily for ease of use. This filter uses an optimized convolution filter with a 3 x 3 kernel with the following weights: [0 1 0 1 -4 1 0 1 0]

The optimized convolution filter used by MPSImageLaplacian can also be used by creating a MPSImageConvolution object with kernelWidth = 3, kernelHeight = 3 and weights as specified above.

4.38.2 Property Documentation

4.38.2.1 - bias [read], [write], [nonatomic], [assign]

The bias is a value to be added to convolved pixel before it is converted back to the storage format. It can be used to convert negative values into a representable range for a unsigned MTLPixelFormat. For example, many edge detection filters produce results in the range [-k,k]. By scaling the filter weights by 0.5/k and adding 0.5, the results will be in range [0,1] suitable for use with unorm formats. It can be used in combination with renormalization of the filter weights to do video ranging as part of the convolution effect. It can also just be used to increase the brightness of the image.

Default value is 0.0f.

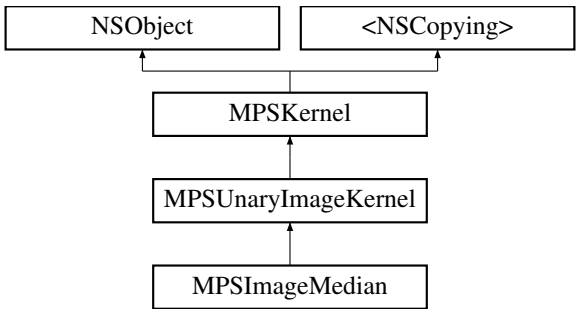
The documentation for this class was generated from the following file:

- MPSImageConvolution.h

4.39 MPSImageMedian Class Reference

`#import <MPSImageMedian.h>`

Inheritance diagram for MPSImageMedian:



Instance Methods

- (nonnull instancetype) - initWithDevice:kernelDiameter:
- (nonnull instancetype) - initWithDevice:

Class Methods

- (NSUInteger) + maxKernelDiameter
- (NSUInteger) + minKernelDiameter

Properties

- NSUInteger kernelDiameter

Additional Inherited Members

4.39.1 Detailed Description

MPSImageMedian.h MetalPerformanceShaders.framework

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved. MetalPerformanceShaders median filters

The MPSImageMedian applies a median filter to an image. A median filter finds the median color value for each channel within a kernelDiameter x kernelDiameter window surrounding the pixel of interest. It is a common means of noise reduction and also as a smoothing filter with edge preserving qualities.

NOTE: The MPSImageMedian filter currently only supports images with <= 8 bits/channel.

4.39.2 Method Documentation

4.39.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.39.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* kernelDiameter:(NSUInteger) *NS_DESIGNATED_INITIALIZER*

Initialize a filter for a particular kernel size and device

Parameters

device	The device the filter will run on
kernelDiameter	Diameter of the median filter. Must be an odd number.

Returns

A valid object or nil, if failure.

4.39.2.3 + (NSUInteger) maxKernelDiameter

The maximum diameter in pixels of the filter window supported by the median filter.

4.39.2.4 + (NSUInteger) minKernelDiameter

The minimum diameter in pixels of the filter window supported by the median filter.

4.39.3 Property Documentation

4.39.3.1 - kernelDiameter [read], [nonatomic], [assign]

The diameter in pixels of the filter window. The median filter is applied to a kernelDiameter x kernelDiameter window of pixels centered on the corresponding source pixel for each destination pixel. The kernel diameter must be an odd number.

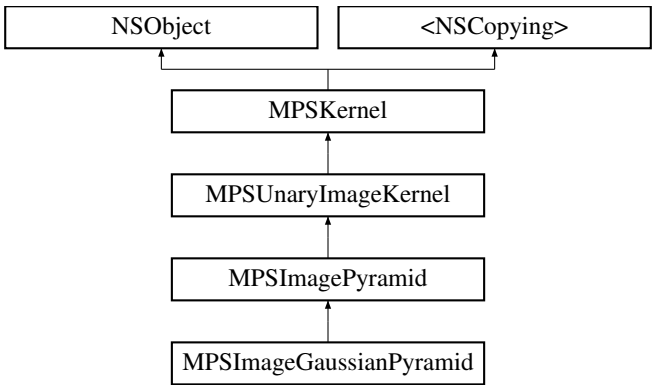
The documentation for this class was generated from the following file:

- MPSImageMedian.h

4.40 MPSImagePyramid Class Reference

`#import <MPSImageConvolution.h>`

Inheritance diagram for MPSImagePyramid:



Instance Methods

- (nonnull instancetype) - initWithDevice:
- (nonnull instancetype) - initWithDevice:centerWeight:
- (nonnull instancetype) - initWithDevice:kernelWidth:kernelHeight:weights:

Properties

- NSUInteger kernelHeight
- NSUInteger kernelWidth

Additional Inherited Members

4.40.1 Detailed Description

The MPSImagePyramid is a base class for creating different kinds of pyramid images

```
Currently supported pyramid-types are:
@ref MPSImageGaussianPyramid

The Gaussian image pyramid kernel is enqueued as a in-place operation using
@ref MPSUnaryImageKernel::encodeToCommandBuffer:inPlaceTexture:fallbackCopyAllocator:
and all mipmap levels after level=1, present in the provided image are filled using
the provided filtering kernel. The fallbackCopyAllocator parameter is not used.

The Gaussian image pyramid filter ignores @ref clipRect and @ref offset and fills
the entire mipmap levels.
```

Note

Make sure your texture type is compatible with mipmapping and supports texture views (see MTLTexture↩ UsagePixelFormatView).
Recall the size of the nth mipmap level:

```
w_n = max(1, floor(w_0 / 2^n))
h_n = max(1, floor(h_0 / 2^n)),
```

where w_0, h_0 are the zeroth level width and height. ie the image dimensions themselves.

4.40.2 Method Documentation

4.40.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device*

Initialize a downwards 5-tap image pyramid with the default filter kernel and device

Parameters

device	The device the filter will run on

The filter kernel is the outer product of w = [1/16, 1/4, 3/8, 1/4, 1/16]^T, with itself

Returns

A valid object or nil, if failure.

Reimplemented from MPSKernel.

4.40.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* centerWeight:(float) *centerWeight*

Initialize a downwards 5-tap image pyramid with a central weight parameter and device

Parameters

device	The device the filter will run on
centerWeight	Defines form of the filter-kernel through the outer product ww^T, where w = [(1/4 - a/2), 1/4, a, 1/4, (1/4 - a/2)]^T and 'a' is centerWeight.

Returns

A valid object or nil, if failure.

4.40.2.3 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* kernelWidth:(NSUInteger) *kernelWidth* kernelHeight:(NSUInteger) *kernelHeight* weights:(const float *__nonnull) *NS_DESIGNATED_INITIALIZER*

Initialize a downwards n-tap pyramid with a custom filter kernel and device

Parameters

device	The device the filter will run on
kernelWidth	The width of the filtering kernel. See MPSImageConvolution.
kernelHeight	The height of the filtering kernel. See MPSImageConvolution.
kernelWeights	A pointer to an array of kernelWidth * kernelHeight values to be used as the kernel. These are in row major order. See MPSImageConvolution.

Returns

A valid object or nil, if failure.

4.40.3 Property Documentation

4.40.3.1 - kernelHeight `[read], [nonatomic], [assign]`

The height of the filter window. Must be an odd number.

4.40.3.2 - kernelWidth `[read], [nonatomic], [assign]`

The width of the filter window. Must be an odd number.

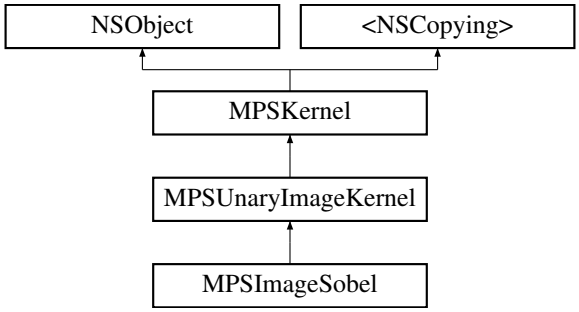
The documentation for this class was generated from the following file:

- MPSImageConvolution.h

4.41 MPSImageSobel Class Reference

`#import <MPSImageConvolution.h>`

Inheritance diagram for MPSImageSobel:



Instance Methods

- (nonnull instancetype) - initWithDevice:
- (nonnull instancetype) - initWithDevice:linearGrayColorTransform:

Properties

- const float * colorTransform

Additional Inherited Members

4.41.1 Detailed Description

The MPSImageSobel implements the Sobel filter. When the color model (e.g. RGB, two-channel, grayscale, etc.) of source and destination textures match, the filter is applied to each channel separately. If the destination is monochrome (single channel) but source multichannel, the pixel values are converted to grayscale before applying Sobel operator using the linear gray color transform vector (v).

```
Luminance = v[0] * pixel.x + v[1] * pixel.y + v[2] * pixel.z;
```

4.41.2 Method Documentation

4.41.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device*

Initialize a Sobel filter on a given device using the default color transform. Default: BT.601/JPEG {0.299f, 0.587f, 0.114f}

For non-default conversion matrices, use -initWithDevice:linearGrayColorTransform:

Parameters

device	The device the filter will run on

Returns

A valid object or nil, if failure.

Reimplemented from MPSKernel.

4.41.2.2 - (nonnull instancetype) **initWithDevice:** (nonnull id< MTLDevice >) *device* linearGrayColorTransform:(const float *__nonnull) *NS_DESIGNATED_INITIALIZER*

Initialize a Sobel filter on a given device with a non-default color transform

Parameters

device	The device the filter will run on
transform	Array of three floats describing the rgb to gray scale color transform. `Luminance = transform[0] * pixel.x + transform[1] * pixel.y + transform`

Returns

A valid object or nil, if failure.

4.41.3 Property Documentation

4.41.3.1 - colorTransform `[read], [nonatomic], [assign]`

Returns a pointer to the array of three floats used to convert RGBA, RGB or RG images to the destination format when the destination is monochrome.

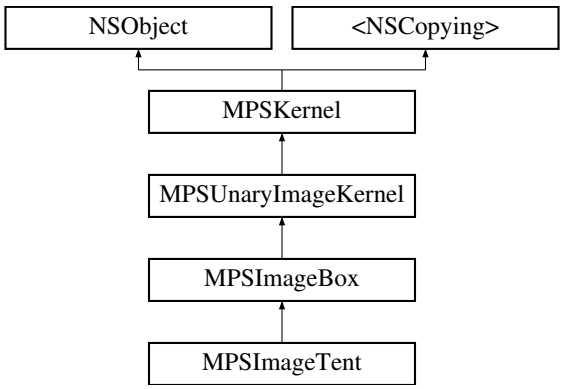
The documentation for this class was generated from the following file:

- MPSImageConvolution.h

4.42 MPSImageTent Class Reference

`#import <MPSImageConvolution.h>`

Inheritance diagram for MPSImageTent:



Additional Inherited Members

4.42.1 Detailed Description

The box filter, while fast, may yield square-ish looking blur effects. However, multiple passes of the box filter tend to smooth out with each additional pass. For example, two 3-wide box blurs produces the same effective convolution as a 5-wide tent blur:

```
1   1   1
    1   1   1
+       1   1   1
================
1   2   3   2   1
```

Addition passes tend to approximate a gaussian line shape.

The MPSImageTent convolves an image with a tent filter. These form a tent shape with incrementally increasing sides, for example:

```
1   2   3   2   1


1   2   1
2   4   2
1   2   1
```

Like the box filter, this arrangement allows for much faster algorithms, espcially for for larger blur radii but with a more pleasing appearance.

The tent blur is a separable filter. The implementation is aware of this and will act accordingly to give best performance for multi-dimensional blurs.

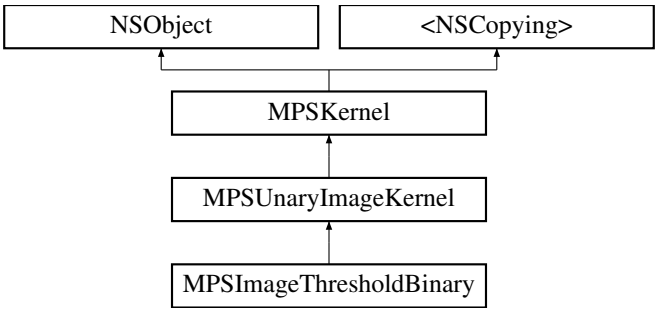
The documentation for this class was generated from the following file:

- MPSImageConvolution.h

4.43 MPSImageThresholdBinary Class Reference

`#import <MPSImageThreshold.h>`

Inheritance diagram for MPSImageThresholdBinary:



Instance Methods

- (nonnull instancetype) - initWithDevice:thresholdValue:maximumValue:linearGrayColorTransform:
- (nonnull instancetype) - initWithDevice:

Properties

- float thresholdValue
- float maximumValue
- const float * transform

Additional Inherited Members

4.43.1 Detailed Description

MPSImageThreshold.h MetalPerformanceShaders

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved. MetalPerformanceShaders thresholding filters

The MPSThreshold filter applies a fixed-level threshold to each pixel in the image. The threshold functions convert a single channel image to a binary image. If the input image is not a single channel image, convert the inputimage to a single channel luminance image using the linearGrayColorTransform and then apply the threshold. The Threshold↩ Binary function is: destinationPixelValue = sourcePixelValue > thresholdValue ? maximumValue : 0

4.43.2 Method Documentation

4.43.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.43.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* thresholdValue:(float) *thresholdValue* maximumValue:(float) *maximumValue* linearGrayColorTransform:(const float *__nullable) *NS_DESIGNATED_INITIALIZER*

initialize a MPSImageThresholdBinary filter

Parameters

device	The device the filter will run on
thresholdValue	The threshold value to use
maximumValue	The maximum value to use
transform	This matrix is an array of 3 floats. The default if no transform is specifed is BT.601/JPEG: {0.299f, 0.587f, 0.114f};

4.43.3 Property Documentation

4.43.3.1 - maximumValue [read], [nonatomic], [assign]

The maximum value used to init the threshold filter

4.43.3.2 - thresholdValue [read], [nonatomic], [assign]

The threshold value used to init the threshold filter

4.43.3.3 - transform [read], [nonatomic], [assign]

The color transform used to init the threshold filter

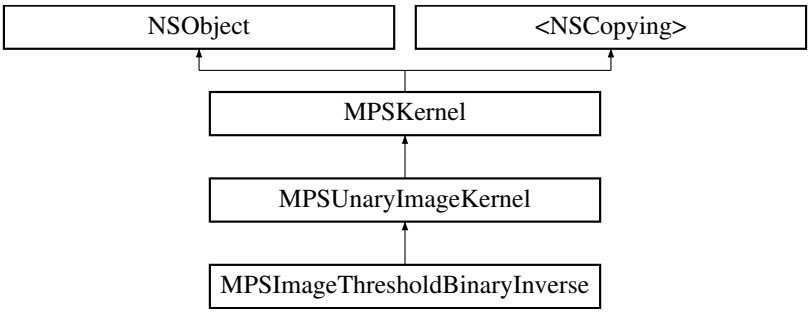
The documentation for this class was generated from the following file:

- MPSImageThreshold.h

4.44 MPSImageThresholdBinaryInverse Class Reference

`#import <MPSImageThreshold.h>`

Inheritance diagram for MPSImageThresholdBinaryInverse:



Instance Methods

- (nonnull instancetype) - initWithDevice:thresholdValue:maximumValue:linearGrayColorTransform:
- (nonnull instancetype) - initWithDevice:

Properties

- float thresholdValue
- float maximumValue
- const float * transform

Additional Inherited Members

4.44.1 Detailed Description

The MPSImageThresholdBinaryInverse filter applies a fixed-level threshold to each pixel in the image. The threshold functions convert a single channel image to a binary image. If the input image is not a single channel image, convert the inputimage to a single channel luminance image using the linearGrayColorTransform and then apply the threshold. The ThresholdBinaryInverse function is: destinationPixelValue = sourcePixelValue > thresholdValue ? 0 : maximumValue

4.44.2 Method Documentation

4.44.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.44.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* thresholdValue:(float) *thresholdValue* maximumValue:(float) *maximumValue* linearGrayColorTransform:(const float *__nullable) *NS_DESIGNATED_INITIALIZER*

initialize a MPSImageThresholdBinaryInverse filter

Parameters

device	The device the filter will run on
thresholdValue	The threshold value to use
maximumValue	The maximum value to use
transform	This matrix is an array of 3 floats. The default if no transform is specifed is BT.601/JPEG: {0.299f, 0.587f, 0.114f};

4.44.3 Property Documentation

4.44.3.1 - maximumValue [read], [nonatomic], [assign]

The maximum value used to init the threshold filter

4.44.3.2 - thresholdValue [read], [nonatomic], [assign]

The threshold value used to init the threshold filter

4.44.3.3 - transform [read], [nonatomic], [assign]

The color transform used to init the threshold filter

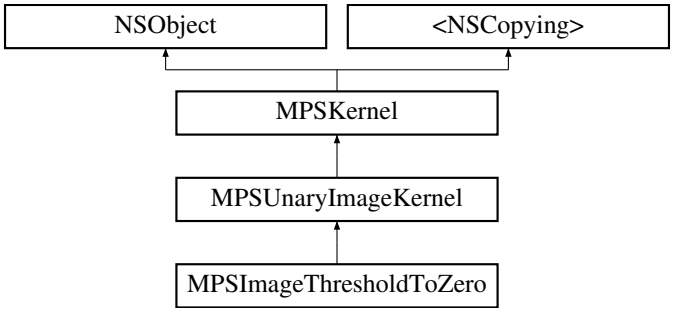
The documentation for this class was generated from the following file:

- MPSImageThreshold.h

4.45 MPSImageThresholdToZero Class Reference

`#import <MPSImageThreshold.h>`

Inheritance diagram for MPSImageThresholdToZero:



Instance Methods

- (nonnull instancetype) - initWithDevice:thresholdValue:linearGrayColorTransform:
- (nonnull instancetype) - initWithDevice:

Properties

- float thresholdValue
- const float * transform

Additional Inherited Members

4.45.1 Detailed Description

The MPSImageThresholdToZero filter applies a fixed-level threshold to each pixel in the image. The threshold functions convert a single channel image to a binary image. If the input image is not a single channel image, convert the inputimage to a single channel luminance image using the linearGrayColorTransform and then apply the threshold. The ThresholdToZero function is: destinationPixelValue = sourcePixelValue > thresholdValue ? sourcePixelValue : 0

4.45.2 Method Documentation

4.45.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.45.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* thresholdValue:(float) *thresholdValue* linearGrayColorTransform:(const float *__nullable) *NS_DESIGNATED_INITIALIZER*

initialize a MPSImageThresholdToZero filter

Parameters

device	The device the filter will run on
thresholdValue	The threshold value to use
transform	This matrix is an array of 3 floats. The default if no transform is specifed is BT.601/JPEG: {0.299f, 0.587f, 0.114f};

4.45.3 Property Documentation

4.45.3.1 - thresholdValue [read], [nonatomic], [assign]

The threshold value used to init the threshold filter

4.45.3.2 - transform [read], [nonatomic], [assign]

The color transform used to init the threshold filter

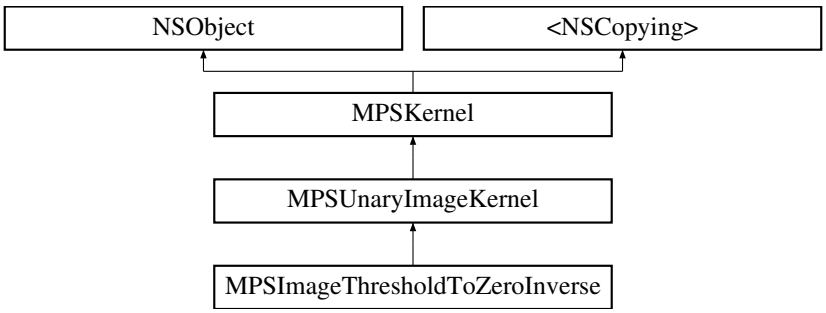
The documentation for this class was generated from the following file:

- MPSImageThreshold.h

4.46 MPSImageThresholdToZeroInverse Class Reference

`#import <MPSImageThreshold.h>`

Inheritance diagram for MPSImageThresholdToZeroInverse:



Instance Methods

- (nonnull instancetype) - initWithDevice:thresholdValue:linearGrayColorTransform:
- (nonnull instancetype) - initWithDevice:

Properties

- float thresholdValue
- const float * transform

Additional Inherited Members

4.46.1 Detailed Description

The MPSImageThresholdToZeroInverse filter applies a fixed-level threshold to each pixel in the image. The threshold functions convert a single channel image to a binary image. If the input image is not a single channel image, convert the inputimage to a single channel luminance image using the linearGrayColorTransform and then apply the threshold. The ThresholdToZeroINverse function is: destinationPixelValue = sourcePixelValue $>$ thresholdValue ? 0 : sourcePixelValue

4.46.2 Method Documentation

4.46.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.46.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* thresholdValue:(float) *thresholdValue* linearGrayColorTransform:(const float *__nullable) *NS_DESIGNATED_INITIALIZER*

initialize a MPSImageThresholdToZeroInverse filter

Parameters

device	The device the filter will run on
thresholdValue	The threshold value to use
transform	This matrix is an array of 3 floats. The default if no transform is specifed is BT.601/JPEG: {0.299f, 0.587f, 0.114f};

4.46.3 Property Documentation

4.46.3.1 - thresholdValue [read], [nonatomic], [assign]

The threshold value used to init the threshold filter

4.46.3.2 - transform [read], [nonatomic], [assign]

The color transform used to init the threshold filter

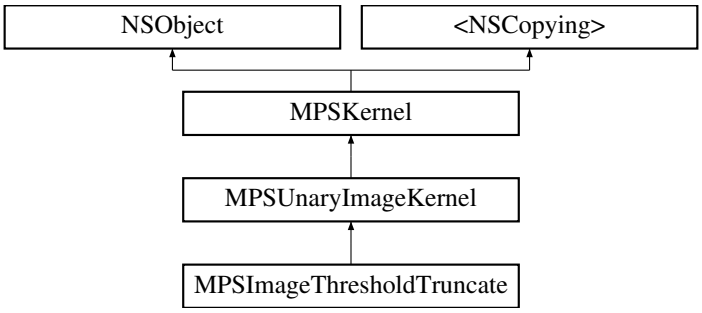
The documentation for this class was generated from the following file:

- MPSImageThreshold.h

4.47 MPSImageThresholdTruncate Class Reference

`#import <MPSImageThreshold.h>`

Inheritance diagram for MPSImageThresholdTruncate:



Instance Methods

- (nonnull instancetype) - initWithDevice:thresholdValue:linearGrayColorTransform:
- (nonnull instancetype) - initWithDevice:

Properties

- float thresholdValue
- const float * transform

Additional Inherited Members

4.47.1 Detailed Description

The MPSImageThresholdTruncate filter applies a fixed-level threshold to each pixel in the image: The threshold functions convert a single channel image to a binary image. If the input image is not a single channel image, convert the inputimage to a single channel luminance image using the linearGrayColorTransform and then apply the threshold. The ThresholdTruncate function is: destinationPixelValue = sourcePixelValue > thresholdValue ? thresholdValue : sourcePixelValue

4.47.2 Method Documentation

4.47.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented from MPSKernel.

4.47.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* thresholdValue:(float) *thresholdValue* linearGrayColorTransform:(const float *__nullable) *NS_DESIGNATED_INITIALIZER*

initialize a MPSImageThresholdTruncate filter

Parameters

device	The device the filter will run on
thresholdValue	The threshold value to use
transform	This matrix is an array of 3 floats. The default if no transform is specifed is BT.601/JPEG: {0.299f, 0.587f, 0.114f};

4.47.3 Property Documentation

4.47.3.1 - thresholdValue [read],[nonatomic],[assign]

The threshold value used to init the threshold filter

4.47.3.2 - transform [read],[nonatomic],[assign]

The color transform used to init the threshold filter

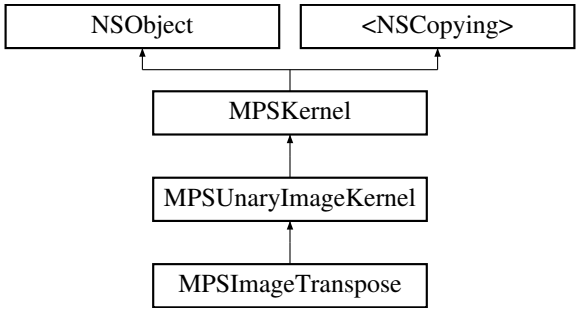
The documentation for this class was generated from the following file:

- MPSImageThreshold.h

4.48 MPSImageTranspose Class Reference

`#import <MPSImageTranspose.h>`

Inheritance diagram for MPSImageTranspose:



Additional Inherited Members

4.48.1 Detailed Description

MPSImageTranspose.h MetalPerformanceShaders.framework

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved. MetalPerformanceShaders transpose filters

The MPSImageTranspose transposes an image

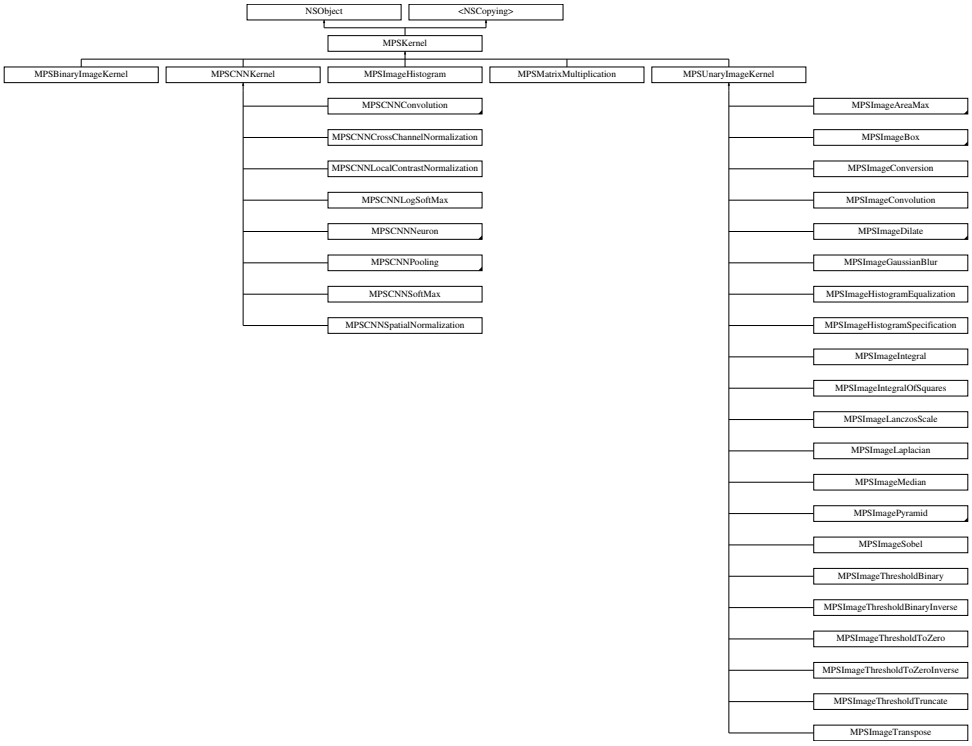
The documentation for this class was generated from the following file:

- MPSImageTranspose.h

4.49 MPSKernel Class Reference

`#import <MPSKernel.h>`

Inheritance diagram for MPSKernel:



Instance Methods

- (nonnull instancetype) - initWithDevice:
- (nonnull instancetype) - copyWithZone:device:
- () - MPSCopyAllocator

Public Attributes

- const MTLRegion MPSRectNoClip

Properties

- MPSKernelOptions options
- id< MTLDevice > device
- NSString * label

4.49.1 Detailed Description

This depends on Metal.framework The MPSKernel class is the base class for all MPS objects. It defines a standard interface for MPS kernels. You should not use the MPSKernel class directly. Instead, a number of MPSKernel subclasses are available in MetalPerformanceShaders.framework that define specific high-performance image processing operations.

The basic sequence for applying a MPSKernel to an image is as follows:

1. Create a MPSKernel corresponding to the operation you wish to perform:

```
MPSImageSobel *sobel = [[MPSImageSobel alloc] initWithDevice: mtlDevice];
```

2. Encode the filter into a command buffer:

```
sobel.offset = ...;
sobel.clipRect = ...;
sobel.options = ...;
[sobel encodeToCommandBuffer: commandBuffer
                sourceTexture: inputImage
           destinationTexture: resultImage ];

if( returnVal < 0 )
    MyShowError( returnVal );
```

   Encoding the kernel merely encodes the operation into a MTLCommandBuffer. It does not modify any pixels, yet. All MPSKernel state has been copied to the command buffer. MPSKernels may be reused. If the texture was previously operated on by another command encoder (e.g. MTLRenderCommandEncoder), you should call -endEncoding on the other encoder before encoding the filter.

   Some MPS filters work in place (inputImage = resultImage) even in situations where Metal might not normally allow in place operation on textures. If in-place operation is desired, you may attempt to call [MPSKernel encodeKernelInPlace...]. If the operation can not be completed in place, then NO will be returned and you will have to create a new result texture and try again. To make an in-place image filter reliable, pass a fallback MPSCopyAllocator to the method to create a new texture to write to in the event that a filter can not operate in place.

   (Repeat steps 2 for more filters, as desired.)

```
It should be self evident that step 2 may not be thread safe. That is, you can not have
multiple threads manipulating the same properties on the same MPSKernel object at the
same time and achieve coherent output. In common usage, the MPSKernel properties don't
often need to be changed from their default values, but if you need to apply the same
filter to multiple images on multiple threads with cropping / tiling, make additional
MPSKernel objects per thread. They are cheap. You can use multiple MPSKernel objects on
multiple threads, as long as only one thread is operating on any particular MPSKernel
object at a time.
```

3. After encoding any additional work to the command buffer using other encoders, submit the MTLCommand↩ Buffer to your MTLCommandQueue, using:

```
[mtlCommandBuffer commit];
```

4.49.2 Method Documentation

4.49.2.1 - (nonnull instancetype) copyWithZone: (nullable NSZone *) *zone* device:(nullable id< MTLDevice >) *device*

Make a copy of this MPSKernel for a new device -copyWithZone: will call this API to make a copy of the MPSKernel on the same device. This interface may also be called directly to make a copy of the MPSKernel on a new device. Typically, the same MPSKernels should not be used to encode kernels on multiple command buffers from multiple threads. Many MPSKernels have mutable properties that might be changed by the other thread while this one is trying to encode. If you need to use a MPSKernel from multiple threads make a copy of it for each additional thread using -copyWithZone: or -copyWithZone:device:

Parameters

zone	The NSZone in which to allocate the object
device	The device for the new MPSKernel. If nil, then use self.device.

Returns

a pointer to a copy of this MPSKernel. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

4.49.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_DESIGNATED_INITIALIZER*

Standard init with default properties per filter type

Parameters

device	The device that the filter will be used on. May not be NULL.

Returns

a pointer to the newly initialized object. This will fail, returning nil if the device is not supported. Devices must be MTLFeatureSet_iOS_GPUFamily2_v1 or later.

Reimplemented in MPSCNNCrossChannelNormalization, MPSCNNLocalContrastNormalization, MPSCNNSpatialNormalization, MPSCNNPooling, MPSCNNConvolution, MPSCNNFullyConnected, MPSImagePyramid, MPSCNNNeuronTanH, MPSCNNNeuronAbsolute, MPSImageSobel, MPSCNNNeuronReLU, MPSCNNNeuronSigmoid, MPSCNNNeuronLinear, MPSImageThresholdToZeroInverse, MPSImageBox, MPSImageGaussianBlur, MPSImageThresholdToZero, MPSImageThresholdTruncate, MPSImageDilate, MPSImageThresholdBinary, MPSMatrixMultiplication, MPSImageThresholdBinaryInverse, MPSImageMedian, and MPSImageAreaMax.

4.49.2.3 - MPSCopyAllocator

A block to make a copy of sourceTexture for MPSKernels that can only execute out of place. Some MPSKernel objects may not be able to operate in place. When that occurs, and in-place operation is requested, MPS will call back to this block to get a new texture to return instead. To avoid spending long periods of time allocating pages to back the MTLTexture, the block should attempt to reuse textures. The texture returned from the MPSCopyAllocator will be returned instead of the sourceTexture from the MPSKernel method on return.

```
// A MPSCopyAllocator to handle cases where in-place operation fails.
MPSCopyAllocator myAllocator = ^id <MTLTexture>( MPSKernel * __nonnull filter,
                                                 __nonnull id <MTLCommandBuffer> cmdBuf,
                                                 __nonnull id <MTLTexture> sourceTexture)
{
    MTLPixelFormat format = sourceTexture.pixelFormat;  // FIXME: is this format writable?
    MTLTextureDescriptor *d = [MTLTextureDescriptor texture2DDescriptorWithPixelFormat: format
                                 width: sourceTexture.width
                                height: sourceTexture.height
                             mipmapped: NO];
    d.usage = MTLTextureUsageShaderRead | MTLTextureUsageShaderWrite;

    //FIXME: Allocating a new texture each time is slow. They take up to 1 ms each.
    //       There are not too many milliseconds in a video frame! You can recycle
    //       old textures (or MTLBuffers and make textures from them) and reuse
    //       the memory here.
    id <MTLTexture> result = [cmdBuf.device newTextureWithDescriptor: d];

    // FIXME: If there is any metadata associated with sourceTexture such as colorspace
    //        information, MTLResource.label, MTLResource.cpuCacheMode mode,
    //        MTLResource.MTLPurgeableState, etc., it may need to be similarly associated
    //        with the new texture to avoid losing your metadata.

    // FIXME: If filter.clipRect doesn't cover the entire image, you may need to copy
    //        pixels from sourceTexture to the new texture or regions of the new texture
    //        will be uninitialized. You can make a MTLCommandEncoder to encode work on
    //        the MTLCommandBuffer here to do that work, if necessary. It will be
    //        scheduled to run immediately before the MPSKernel work. Do not call
    //        [MTLCommandBuffer enqueue/commit/waitUntilCompleted/waitUntilScheduled]
    //        in the MPSCopyAllocator block. Make sure to call -endEncoding on the
    //        MTLCommandEncoder so that the MTLCommandBuffer has no active encoder
    //        before returning.

    // CAUTION: The next command placed on the MTLCommandBuffer after the MPSCopyAllocator
    //          returns is almost assuredly going to be encoded with a MTLComputeCommandEncoder.
    //          Creating any other type of encoder in the MPSCopyAllocator will probably cost
    //          an additional 0.5 ms of both CPU _AND_ GPU time (or more!) due to a double
    //          mode switch penalty.

    // CAUTION: If other objects (in addition to the caller of -encodeToCommandBuffer:inPlaceTexture:...)
    //          own a reference to sourceTexture, they may need to be notified that
    //          sourceTexture has been replaced so that they can release that resource
    //          and adopt the new texture.

    //          The reference to sourceTexture owned by the caller of
    //          -encodeToCommandBuffer:inPlaceTexture... will be released by
    //          -encodeToCommandBuffer:inPlaceTexture:... after the kernel is encoded if
    //          and only if the MPSCopyAllocator is called, and the operation is successfully
    //          encoded out of place.

    return result;
```

```
    // d is autoreleased
};
```

If nil is returned by the allocator, NO will be returned by the calling function.

When the MPSCopyAllocator is called, no MTLCommandEncoder is active on the commandBuffer. You may create a MTLCommandEncoder in the block to initialize the texture. Make sure to call -endEncoding on it before returning, if you do.

Parameters

filter	A valid pointer to the MPSKernel that is calling the MPSCopyAllocator. From it you can get the clipRect of the intended operation.
commandBuffer	A valid MTLCommandBuffer. It can be used to obtain the device against which to allocate the new texture. You may also enqueue operations on the commandBuffer to initialize the texture on a encoder allocated in the block. You may not submit, enqueue or wait for scheduling/completion of the command buffer.
sourceTexture	The texture that is providing the source image for the filter. You may wish to use its size and MTLPixelFormat for the new texture, but it is not requred.

Returns

A new valid MTLTexture to use as the destination for the MPSKernel. If the calling function succeeds, its texture parameter will be overwritten with a pointer to this texture. If the calling function fails (highly unlikely, except for user error) then the texture will be released before the calling function returns.

4.49.3 Member Data Documentation

4.49.3.1 - (const MTLRegion) MPSRectNoClip

MPSRectNoClip This is a special constant to indicate no clipping is to be done. The entire image will be used. This is the default clipping rectangle or the input extent for MPSKernels.

4.49.4 Property Documentation

4.49.4.1 - device `[read],[nonatomic],[retain]`

The device on which the kernel will be used

4.49.4.2 - label `[read],[write],[atomic],[copy]`

A string to help identify this object.

4.49.4.3 - options `[read],[write],[nonatomic],[assign]`

The set of options used to run the kernel. MPSKernelOptions

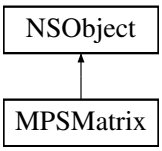
The documentation for this class was generated from the following files:

- MPSKernel.h
- MPSTypes.h

4.50 MPSMatrix Class Reference

```
#import <MPSMatrix.h>
```

Inheritance diagram for MPSMatrix:



Instance Methods

- (nonnull instancetype) - initWithBuffer:descriptor:
- (nonnull instancetype) - **NS_UNAVAILABLE**

Properties

- id< MTLDevice > device
- NSUInteger rows
- NSUInteger columns
- MPSDataType dataType
- NSUInteger rowBytes
- id< MTLBuffer > data

4.50.1 Detailed Description

This depends on Metal.framework

A MPSMatrix object describes a 2-dimensional array of data and provides storage for its values. MPSMatrix objects serve as inputs and outputs of MPSMatrixKernel objects.

Implementation note: A MPSMatrix object maintains its internal storage using a MTLBuffer object and thus the same rules for maintaining coherency of a MTLBuffer's data between CPU memory and GPU memory apply to a MPSMatrix. Data is assumed to be stored in row-major layout.

4.50.2 Method Documentation

4.50.2.1 - (nonnull instancetype) initWithBuffer: (nonnull id< MTLBuffer >) *buffer* descriptor:(nonnull MPSMatrixDescriptor *) *descriptor*

Initialize a MPSMatrix object with a MTLBuffer.

Parameters

buffer	The MTLBuffer object which contains the data to use for the MPSMatrix. May not be NULL.
descriptor	The MPSMatrixDescriptor. May not be NULL.

Returns

A valid MPSMatrix object or nil, if failure.

This function returns a MPSMatrix object which uses the supplied MTLBuffer. The dimensions and stride of the matrix are specified by the MPSMatrixDescriptor object.

The provided MTLBuffer must have enough storage to hold

```
(descriptor.rows-1) * descriptor.rowBytes + descriptor.columns * (element size) bytes.
```

4.50.3 Property Documentation

4.50.3.1 - columns [read], [nonatomic], [assign]

The number of columns in the MPSMatrix.

4.50.3.2 - data [read], [nonatomic], [assign]

An MTLBuffer to store the data.

4.50.3.3 - dataType [read], [nonatomic], [assign]

The type of the MPSMatrix data.

4.50.3.4 - device [read], [nonatomic], [retain]

The device on which the MPSMatrix will be used.

4.50.3.5 - rowBytes [read], [nonatomic], [assign]

The stride, in bytes, between corresponding elements of consecutive rows.

4.50.3.6 - rows [read], [nonatomic], [assign]

The number of rows in the MPSMatrix.

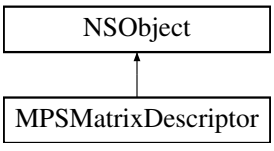
The documentation for this class was generated from the following file:

- MPSMatrix.h

4.51 MPSMatrixDescriptor Class Reference

`#import <MPSMatrix.h>`

Inheritance diagram for MPSMatrixDescriptor:



Class Methods

- (__nonnull instancetype) + matrixDescriptorWithDimensions:columns:rowBytes:dataType:
- (size_t) + rowBytesFromColumns:dataType:

Properties

- NSUInteger rows
- NSUInteger columns

- MPSDataType dataType
- NSUInteger rowBytes

4.51.1 Detailed Description

This depends on Metal.framework

A MPSMatrixDescriptor describes the sizes, strides, and data type of a 2-dimensional array of data. All storage is assumed to be in row-major order.

4.51.2 Method Documentation

4.51.2.1 + (__nonnull instancetype) matrixDescriptorWithDimensions: (NSUInteger) *rows* columns:(NSUInteger) *columns* rowBytes:(NSUInteger) *rowBytes* dataType:(MPSDataType) *dataType*

Create a MPSMatrixDescriptor with the specified dimensions and data type.

Parameters

rows	The number of rows of the matrix.
columns	The number of columns of the matrix.
rowBytes	The number of bytes between starting elements of consecutive rows. Must be a multiple of the element size.
dataType	The type of the data to be stored in the matrix.

For performance considerations the optimal row stride may not necessarily be equal to the number of columns in the matrix. The MPSMatrix class provides a method which may be used to determine this value, see the rowBytesFromColumns API in the MPSMatrix class.

4.51.2.2 + (size_t) rowBytesFromColumns: (NSUInteger) *columns* dataType:(MPSDataType) *dataType*

Return the recommended row stride, in bytes, for a given number of columns.

Parameters

columns	The number of columns in the matrix for which the recommended row stride, in bytes, is to be determined.
dataType	The type of matrix data values.

To achieve best performance the optimal stride between rows of a matrix is not necessarily equivalent to the number of columns. This method returns the row stride, in bytes, which gives best performance for a given number of columns. Using this row stride to construct your array is recommended, but not required (provided that the stride used is still large enough to allocate a full row of data).

4.51.3 Property Documentation

4.51.3.1 - columns [read], [write], [nonatomic], [assign]

The number of columns in the matrix.

4.51.3.2 - dataType [read], [write], [nonatomic], [assign]

The type of the data which makes up the values of the matrix.

4.51.3.3 - rowBytes [read], [write], [nonatomic], [assign]

The stride, in bytes, between corresponding elements of consecutive rows. Must be a multiple of the element size.

4.51.3.4 - rows [read],[write],[nonatomic],[assign]

The number of rows in the matrix.

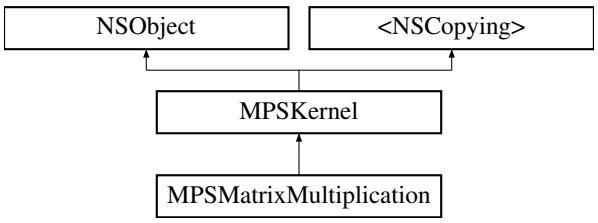
The documentation for this class was generated from the following file:

- MPSMatrix.h

4.52 MPSMatrixMultiplication Class Reference

`#import <MPSMatrixMultiplication.h>`

Inheritance diagram for MPSMatrixMultiplication:



Instance Methods

- (nonnull instancetype) - initWithDevice:transposeLeft:transposeRight:resultRows:resultColumns:interior↩ Columns:alpha:beta:
- (nonnull instancetype) - initWithDevice:
- (void) - encodeToCommandBuffer:leftMatrix:rightMatrix:resultMatrix:

Properties

- MTLOrigin resultMatrixOrigin
- MTLOrigin leftMatrixOrigin
- MTLOrigin rightMatrixOrigin

Additional Inherited Members

4.52.1 Detailed Description

MPSMatrixMultiplication.h MetalPerformanceShaders.framework

Copyright

Copyright (c) 2016 Apple Inc. All rights reserved. MetalPerformanceShaders filter base classes

This depends on Metal.framework.

A matrix multiplication kernel.

A MPSMatrixMultiplication object computes:

```
C = alpha * op(A) * op(B) + beta * C

A, B, and C are matrices which are represented by MPSMatrix
objects. alpha and beta are scalar values (of the same data type
```

```
as values of C) which are applied as shown above.  A and B may
each have an optional transposition operation applied.

A, B, and C (also referred to in later discussions as the left input
matrix, the right input matrix, and the result matrix respectively).

A MPSMatrixMultiplication object is initialized with the transpose
operators to apply to A and B, sizes for the operation to perform,
and the scalar values alpha and beta.
```

4.52.2 Method Documentation

4.52.2.1 - (void) encodeToCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* leftMatrix:(MPSMatrix const *__nonnull) *leftMatrix* rightMatrix:(MPSMatrix const *__nonnull) *rightMatrix* resultMatrix:(encode(commandBuffer:leftMatrix:rightMatrix:resultMatrix:)) *MPS_SWIFT_NAME*

Encode a MPSMatrixMultiplication object to a command buffer.

Parameters

commandBuffer	A valid MTLCommandBuffer to receive the encoded kernel.
leftMatrix	A valid MPSMatrix object which specifies the left input matrix.
rightMatrix	A valid MPSMatrix object which specifies the right input matrix.
resultMatrix	A valid MPSMatrix object which specifies the addend matrix which will also be overwritten by the result.

Certain constraints apply to the sizes of the matrices depending on the transposition operations and sizes requested at initialization time as well as the origins at the time this routine is called:

The left input matrix must be large enough to hold an array of size resultRows x interiorColumns elements beginning at leftMatrixOrigin.

The right input matrix must be large enough to hold an array of size interiorColumns x resultColumns elements beginning at rightMatrixOrigin.

The result matrix must be large enough to hold an array of size resultRows x resultColumns elements beginning at resultMatrixOrigin.

4.52.2.2 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *NS_UNAVAILABLE*

Use the above initialization method instead.

Reimplemented from MPSKernel.

4.52.2.3 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* transposeLeft:(BOOL) *transposeLeft* transposeRight:(BOOL) *transposeRight* resultRows:(NSUInteger) *resultRows* resultColumns:(NSUInteger) *resultColumns* interiorColumns:(NSUInteger) *interiorColumns* alpha:(double) *alpha* beta:(double) *beta*

Initialize an MPSMatrixMultiplication object on a device for a given size and desired transpose and scale values.

Parameters

device	The device on which the kernel will execute.
transposeLeft	A boolean value which indicates if the left input matrix should be used in transposed form. If 'YES' then op(A) = A**T, otherwise op(A) = A.
transposeRight	A boolean value which indicates if the right input matrix should be used in transposed form. If 'YES' then op(B) = B**T, otherwise op(B) = B.

resultRows	The number of rows in the result matrix, M in BLAS GEMM description.
resultColumns	The number of columns in the result matrix, N in BLAS GEMM description.
interiorColumns	The number of columns of the left input matrix after the appropriate transpose operation has been applied. K in BLAS GEMM description.
alpha	The scale factor to apply to the product. Specified in double precision. Will be converted to the appropriate precision in the implementation subject to rounding and/or clamping as necessary.
beta	The scale factor to apply to the initial values of C. Specified in double precision. Will be converted to the appropriate precision in the implementation subject to rounding and/or clamping as necessary.

Returns

A valid MPSMatrixMultiplication object or nil, if failure.

4.52.3 Property Documentation

4.52.3.1 - leftMatrixOrigin `[read], [write], [nonatomic], [assign]`

The origin, relative to [0, 0] in the left input matrix, at which to start reading values. This property is modifiable and defaults to [0, 0] at initialization time. If a different origin is desired then this should be modified prior to encoding the kernel. The z value must be 0.

4.52.3.2 - resultMatrixOrigin `[read], [write], [nonatomic], [assign]`

The origin, relative to [0, 0] in the result matrix, at which to start writing (and reading if necessary) results. This property is modifiable and defaults to [0, 0] at initialization time. If a different origin is desired then this should be modified prior to encoding the kernel. The z value must be 0.

4.52.3.3 - rightMatrixOrigin `[read], [write], [nonatomic], [assign]`

The origin, relative to [0, 0] in the right input matrix, at which to start reading values. This property is modifiable and defaults to [0, 0] at initialization time. If a different origin is desired then this should be modified prior to encoding the kernel. The z value must be 0.

The documentation for this class was generated from the following file:

- MPSMatrixMultiplication.h

4.53 MPSOffset Struct Reference

`#include <MPSTypes.h>`

Public Attributes

- NSInteger x
- NSInteger y
- NSInteger z

4.53.1 Detailed Description

A signed coordinate with x, y and z components

4.53.2 Member Data Documentation

4.53.2.1 NSInteger MPSOffset::x

The horizontal component of the offset. Units: pixels

4.53.2.2 NSInteger MPSOffset::y

The vertical component of the offset. Units: pixels

4.53.2.3 NSInteger MPSOffset::z

The depth component of the offset. Units: pixels

The documentation for this struct was generated from the following file:

- MPSTypes.h

4.54 MPSOrigin Struct Reference

`#include <MPSTypes.h>`

Public Attributes

- double x
- double y
- double z

4.54.1 Detailed Description

A position in an image

4.54.2 Member Data Documentation

4.54.2.1 double MPSOrigin::x

The x coordinate of the position

4.54.2.2 double MPSOrigin::y

The y coordinate of the position

4.54.2.3 double MPSOrigin::z

The z coordinate of the position

The documentation for this struct was generated from the following file:

- MPSTypes.h

4.55 MPSRegion Struct Reference

`#include <MPSTypes.h>`

Public Attributes

- MPSOrigin origin
- MPSSize size

4.55.1 Detailed Description

A region of an image

4.55.2 Member Data Documentation

4.55.2.1 MPSOrigin MPSRegion::origin

The top left corner of the region. Units: pixels

4.55.2.2 MPSSize MPSRegion::size

The size of the region. Units: pixels

The documentation for this struct was generated from the following file:

- MPSTypes.h

4.56 MPSScaleTransform Struct Reference

`#include <MPSTypes.h>`

Public Attributes

- double **scaleX**
- double **scaleY**
- double **translateX**
- double **translateY**

4.56.1 Detailed Description

Transform matrix for explict control over resampling in MPSImageLanczosScale. The MPSScaleTransform is equivalent to:

```
(CGAffineTransform) {
     .a = scaleX,         .b = 0,
     .c = 0,              .d = scaleY,
     .tx = translateX,    .ty = translateY
 }
```

The documentation for this struct was generated from the following file:

- MPSTypes.h

4.57 MPSSize Struct Reference

`#include <MPSTypes.h>`

Public Attributes

- double width
- double height
- double depth

4.57.1 Detailed Description

A size of a region in an image

4.57.2 Member Data Documentation

4.57.2.1 double MPSSize::depth

The depth of the region

4.57.2.2 double MPSSize::height

The height of the region

4.57.2.3 double MPSSize::width

The width of the region

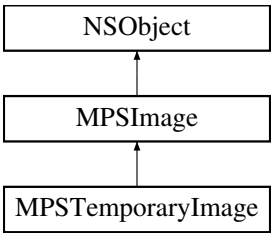
The documentation for this struct was generated from the following file:

- MPSTypes.h

4.58 MPSTemporaryImage Class Reference

`#import <MPSImage.h>`

Inheritance diagram for MPSTemporaryImage:



Instance Methods

- (nonnull instancetype) - initWithTexture:featureChannels:
- (nonnull instancetype) - initWithDevice:imageDescriptor:

Class Methods

- (nonnull instancetype) + temporaryImageWithCommandBuffer:imageDescriptor:
- (nonnull instancetype) + temporaryImageWithCommandBuffer:textureDescriptor:
- (void) + prefetchStorageWithCommandBuffer:imageDescriptorList:

Properties

- NSUInteger **readCount**

4.58.1 Detailed Description

MPSImage MPSTemporaryImages are provided as a fast way to store transient data that will be used promptly and discarded.

MPSTemporaryImages can provide for profound reduction in the aggregate texture memory and associated CP↩ U-side allocation cost in your application. MPS achieves this by automatically identifying MPSTemporaryImages that do not overlap in time over the course of a MTLCommandBuffer and so can reuse the same memory. M↩ PSTemporaryImages leverage MPS's internal cache of preallocated reusable memory to hold pixel data to avoid typical memory allocation performance penalties common to ordinary MPSImages and MTLTextures.

To avoid data corruption due to aliasing, MPSTemporaryImages impose some important restrictions:

- The textures are MTLStorageModePrivate. You can not, for example, use [MTLTexture getBytes...] or [MT↩ LTexture replaceRegion...] with them. MPSTemporaryImages are strictly read and written by the GPU.
- The temporary image may be used only on a single MTLCommandBuffer. This limits the chronology to a single linear time stream.
- The readCount property must be managed correctly. Please see the description of the readCount property for full details.
- see also pixel format restrictions for MPSImages in general.

Since MPSTemporaryImages can only be used with a single MTLCommandBuffer, and can not be used off the GPU, they generally should not be kept around past the completion of the MTLCommandBuffer. The lifetime of MPSTemporaryImages is expected to be typically extremely short, perhaps only a few lines of code.

To keep the lifetime of the underlying texture allocation as short as possible, the underlying texture is not allocated until the first time the MPSTemporaryImage is used by a MPSCNNKernel or the .texture property is read. The readCount property serves to limit the lifetime on the other end.

You may use the MPSTemporaryImage.texture with MPSUnaryImageKernel -encode... methods, iff feature↩ Channels <= 4 and the MTLTexture conforms to requirements of that MPSKernel. In such cases, the readCount is not modified, since the enclosing object is not available. There is no locking mechanism provided to prevent a MTLTexture returned from the .texture property from becoming invalid when the readCount reaches 0.

MPSTemporaryImages can otherwise be used wherever MPSImages are used.

4.58.2 Method Documentation

4.58.2.1 - (nonnull instancetype) initWithDevice: (nonnull id< MTLDevice >) *device* imageDescriptor:(const MPSImageDescriptor *__nonnull) *NS_UNAVAILABLE*

Unavailable. Use itemporaryImageForCommandBuffer:textureDescriptor: instead.

Reimplemented from MPSImage.

4.58.2.2 - (nonnull instancetype) initWithTexture: (nonnull id< MTLTexture >) *texture* featureChannels:(NSUInteger) *NS_UNAVAILABLE*

Unavailable. Use temporaryImageForCommandBuffer:textureDescriptor: or -temporaryImageForCommandBuffer:imageDescriptor: instead.

Reimplemented from MPSImage.

4.58.2.3 + (void) prefetchStorageWithCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* imageDescriptorList:(NSArray< **MPSImageDescriptor** * > *__nonnull) *descriptorList*

Help MPS decide which allocations to make ahead of time The texture cache that underlies the MPSTemporaryImage can automatically allocate new storage as needed as you create new temporary images. However, sometimes a more global view of what you plan to make is useful for maximizing memory reuse to get the most efficient operation. This class method hints to the cache what the list of images will be.

It is never necessary to call this method. It is purely a performance and memory optimization.

Parameters

commandBuffer	The command buffer on which the MPSTemporaryImages will be used
descriptorList	A NSArray of MPSImageDescriptors, indicating images that will be created

4.58.2.4 + (nonnull instancetype) temporaryImageWithCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* imageDescriptor:(const **MPSImageDescriptor** *__nonnull) *imageDescriptor*

Initialize a MPSTemporaryImage for use on a MTLCommandBuffer

Parameters

commandBuffer	The MTLCommandBuffer on which the MPSTemporaryImage will be exclusively used
imageDescriptor	A valid imageDescriptor describing the MPSImage format to create.

Returns

A valid MPSTemporaryImage. The object will be released when the command buffer is committed. The underlying texture will become invalid before this time due to the action of the readCount property.

4.58.2.5 + (nonnull instancetype) temporaryImageWithCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* textureDescriptor:(const MTLTextureDescriptor *__nonnull) *textureDescriptor*

Low level interface for creating a MPSTemporaryImage using a MTLTextureDescriptor This function provides access to MTLPixelFormats not typically covered by -initForCommandBuffer:imageDescriptor: The feature channels will be inferred from the MTLPixelFormat without changing the width. The following restrictions apply:

```
MTLTextureType must be MTLTextureType2D or MTLTextureType2DArray
MTLTextureUsage must contain at least one of MTLTextureUsageShaderRead, MTLTextureUsageShaderWrite
MTLStorageMode must be MTLStorageModePrivate
depth must be 1
```

Parameters

commandBuffer	The command buffer on which the MPSTemporaryImage may be used
textureDescriptor	A texture descriptor describing the MPSTemporaryImage texture

Returns

A valid MPSTemporaryImage. The object will be released when the command buffer is committed. The underlying texture will become invalid before this time due to the action of the readCount property.

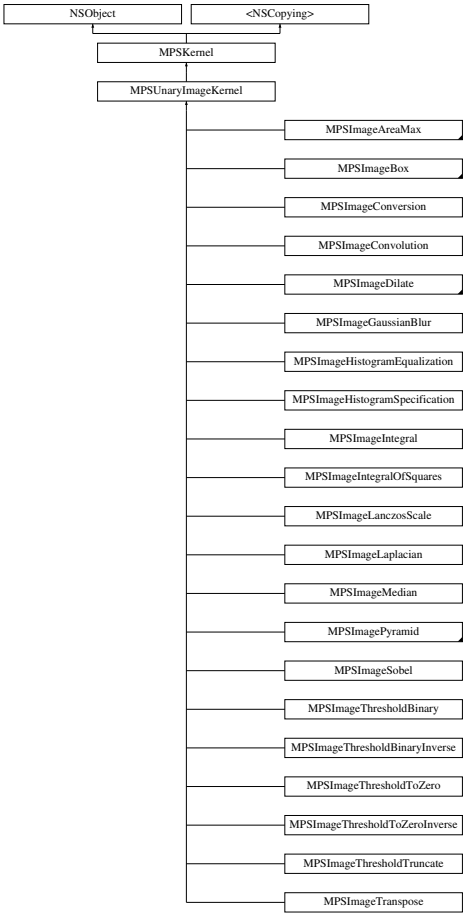
The documentation for this class was generated from the following file:

- MPSImage.h

4.59 MPSUnaryImageKernel Class Reference

`#import <MPSImageKernel.h>`

Inheritance diagram for MPSUnaryImageKernel:



Instance Methods

- (BOOL) - encodeToCommandBuffer:inPlaceTexture:fallbackCopyAllocator:
- (void) - encodeToCommandBuffer:sourceTexture:destinationTexture:
- (MPSRegion) - sourceRegionForDestinationSize:

Properties

- MPSOffset offset
- MTLRegion clipRect
- MPSImageEdgeMode edgeMode

Additional Inherited Members

4.59.1 Detailed Description

MPSImageKernel.h MetalPerformanceShaders.framework

Copyright

Copyright (c) 2015 Apple Inc. All rights reserved. MetalPerformanceShaders filter base classes

This depends on Metal.framework A MPSUnaryImageKernel consumes one MTLTexture and produces one MTL↩ Texture.

4.59.2 Method Documentation

4.59.2.1 - (BOOL) encodeToCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* inPlaceTexture:(__nonnull id< MTLTexture > __strong *__nonnull) *texture* fallbackCopyAllocator:(encode(commandBuffer:inPlaceTexture↩ :fallbackCopyAllocator:)) *MPS_SWIFT_NAME*

This method attempts to apply the MPSKernel in place on a texture.

```
In-place operation means that the same texture is used both to hold the input
image and the results. Operating in-place can be an excellent way to reduce
resource utilization, and save time and energy. While simple Metal kernels can
not operate in place because textures can not be readable and writable at the
same time, some MPSKernels can operate in place because they use
multi-pass algorithms. Whether a MPSKernel can operate in-place can
depend on current hardware, operating system revision and the parameters
and properties passed to it. You should never assume that a MPSKernel will
continue to work in place, even if you have observed it doing so before.
```

If the operation succeeds in-place, YES is returned. If the in-place operation fails and no copyAllocator is provided, then NO is returned. Without a fallback MPSCopyAllocator, in neither case is the pointer held at *texture modified.

Failure during in-place operation is very common and will occur inconsistently across different hardware platforms and OS releases. Without a fallback MPSCopyAllocator, operating in place may require significant error handling code to accompany each call to -encodeToCommandBuffer:..., complicating your code.

You may find it simplifies your code to provide a fallback MPSCopyAllocator so that the operation can proceed reliably even when it can not proceed in-place. When an in-place filter fails, the MPSCopyAllocator (if any) will be invoked to create a new texture in which to write the results, allowing the filter to proceed reliably out-of-place. The original texture will be released, replaced with a pointer to the new texture and YES will be returned. If the allocator returns an invalid texture, it is released, *texture remains unmodified and NO is returned. Please see the MPSCopyAllocator definition for a sample allocator implementation.

Sample usage with a copy allocator:

```
id <MTLTexture> inPlaceTex = ...;
MPSImageSobel *sobelFiler = [[MPSImageSobel alloc] initWithDevice: my_device];

// With a fallback MPSCopyAllocator, failure should only occur in exceptional
// conditions such as MTLTexture allocation failure or programmer error.
// That is, the operation is roughly as robust as the MPSCopyAllocator.
// Depending on the quality of that, we might decide we are justified here
// in not checking the return value.
[sobelFilter encodeToCommandBuffer: my_command_buffer
                     inPlaceTexture: &inPlaceTex  // may be replaced!
               fallbackCopyAllocator: myAllocator];

// If myAllocator was not called:
//
//      inPlaceTex holds the original texture with the result pixels in it
//
// else,
//
//      1) myAllocator creates a new texture.
//      2) The new texture pixel data is overwritten by MPSUnaryImageKernel.
```

```
//      3) The old texture passed in *inPlaceTex is released once.
//      4) *inPlaceTex = the new texture
//
// In either case, the caller should now hold one reference to the texture now held in
// inPlaceTex, whether it was replaced or not. Most of the time that means that nothing
// further needs to be done here, and you can proceed to the next image encoding operation.
// However, if other agents held references to the original texture, they still hold them
// and may need to be alerted that the texture has been replaced so that they can retain
// the new texture and release the old one.

[sobelFilter release];  // if not ARC, clean up the MPSImageSobel object
```

Note: Image filters that look at neighboring pixel values may actually consume more memory when operating in place than out of place. Many such operations are tiled internally to save intermediate texture storage, but can not tile when operating in place. The memory savings for tiling is however very short term, typically the lifetime of the MTLCommandBuffer.

Attempt to apply a MPSKernel to a texture in place.

Parameters

commandBuffer	A valid MTLCommandBuffer to receive the encoded filter
texture	A pointer to a valid MTLTexture containing source image. On success, the image contents and possibly texture itself will be replaced with the result image.
copyAllocator	An optional block to allocate a new texture to hold the results, in case in-place operation is not possible. The allocator may use a different MTLPixelFormat or size than the original texture. You may enqueue operations on the provided MTLCommandBuffer using the provided MTLComputeCommandEncoder to initialize the texture contents.

Returns

On success, YES is returned. The texture may have been replaced with a new texture if a copyAllocator was provided. On failure, NO is returned. The texture is unmodified.

4.59.2.2 - (void) encodeToCommandBuffer: (nonnull id< MTLCommandBuffer >) *commandBuffer* sourceTexture:(nonnull id< MTLTexture >) *sourceTexture* destinationTexture:(encode(commandBuffer:sourceTexture:destinationTexture:)) *MPS_SWIFT_NAME*

Encode a MPSKernel into a command Buffer. The operation shall proceed out-of-place.

Parameters

commandBuffer	A valid MTLCommandBuffer to receive the encoded filter
sourceTexture	A valid MTLTexture containing the source image.
destinationTexture	A valid MTLTexture to be overwritten by result image. DestinationTexture may not alias sourceTexture.

4.59.2.3 - (MPSRegion) sourceRegionForDestinationSize: (sourceRegion(destinationSize:)) *MPS_SWIFT_NAME*

sourceRegionForDestinationSize: is used to determine which region of the sourceTexture will be read by encodeToCommandBuffer:sourceTexture:destinationTexture (and similar) when the filter runs. This information may be needed if the source image is broken into multiple textures. The size of the full (untiled) destination image is provided. The region of the full (untiled) source image that will be read is returned. You can then piece together an appropriate texture containing that information for use in your tiled context.

The function will consult the MPSUnaryImageKernel offset and clipRect parameters, to determine the full region read by the function. Other parameters such as sourceClipRect, kernelHeight and kernelWidth will be consulted as necessary. All properties should be set to intended values prior to calling sourceRegionForDestinationSize:.

```
Caution: This function operates using global image coordinates, but
-encodeToCommandBuffer:... uses coordinates local to the source and
destination image textures. Consequently, the offset and clipRect
attached to this object will need to be updated using a global to
```

```
local coordinate transform before -encodeToCommandBuffer:... is
called.
```

Determine the region of the source texture that will be read for a encode operation

Parameters

destinationSize	The size of the full virtual destination image.

Returns

The area in the virtual source image that will be read.

4.59.3 Property Documentation

4.59.3.1 - clipRect [read], [write], [nonatomic], [assign]

An optional clip rectangle to use when writing data. Only the pixels in the rectangle will be overwritten. A MTL↩ Region that indicates which part of the destination to overwrite. If the clipRect does not lie completely within the destination image, the intersection between clip rectangle and destination bounds is used. Default: MPSRectNoClip (MPSKernel::MPSRectNoClip) indicating the entire image.

See Also: MPSKernel clipRect

4.59.3.2 - edgeMode [read], [write], [nonatomic], [assign]

The MPSImageEdgeMode to use when texture reads stray off the edge of an image Most MPSKernel objects can read off the edge of the source image. This can happen because of a negative offset property, because the offset + clipRect.size is larger than the source image or because the filter looks at neighboring pixels, such as a Convolution or morphology filter. Default: usually MPSImageEdgeModeZero. (Some MPSKernel types default to MPSImageEdgeModeClamp, because MPSImageEdgeModeZero is either not supported or would produce unexpected results.)

See Also: MPSKernelEdgeMode

4.59.3.3 - offset [read], [write], [nonatomic], [assign]

The position of the destination clip rectangle origin relative to the source buffer. The offset is defined to be the position of clipRect.origin in source coordinates. Default: {0,0,0}, indicating that the top left corners of the clipRect and source image align.

See Also: subsubsection_mpsoffset

The documentation for this class was generated from the following file:

- MPSImageKernel.h

Index

alpha
MPSCNNCrossChannelNormalization, 29
MPSCNNLocalContrastNormalization, 37
MPSCNNSpatialNormalization, 53

beta
MPSCNNCrossChannelNormalization, 29
MPSCNNLocalContrastNormalization, 37
MPSCNNSpatialNormalization, 53
bias
MPSImageConvolution, 65
MPSImageLaplacian, 83

channelFormat
MPSImageDescriptor, 67
clipRect
MPSBinaryImageKernel, 20
MPSCNNKernel, 34
MPSUnaryImageKernel, 119
clipRectSource
MPSImageHistogram, 74
cnnConvolutionDescriptorWithKernelWidth:kernel↩
Height:inputFeatureChannels:output↩
FeatureChannels:neuronFilter:
MPSCNNConvolutionDescriptor, 25
colorTransform
MPSImageSobel, 88
columns
MPSMatrix, 106
MPSMatrixDescriptor, 107
copyWithZone:device:
MPSKernel, 101
cpuCacheMode
MPSImageDescriptor, 67

data
MPSMatrix, 106
dataType
MPSMatrix, 106
MPSMatrixDescriptor, 107
delta
MPSCNNCrossChannelNormalization, 29
MPSCNNLocalContrastNormalization, 37
MPSCNNSpatialNormalization, 53
depth
MPSSize, 113
destinationAlpha
MPSImageConversion, 63
destinationFeatureChannelOffset
MPSCNNKernel, 34

device
MPSImage, 56
MPSKernel, 104
MPSMatrix, 106

edgeMode
MPSCNNKernel, 35
MPSUnaryImageKernel, 119
encodeToCommandBuffer:inPlacePrimaryTexture↩
:secondaryTexture:fallbackCopyAllocator:
MPSBinaryImageKernel, 18
encodeToCommandBuffer:inPlaceTexture:fallback↩
CopyAllocator:
MPSUnaryImageKernel, 117
encodeToCommandBuffer:leftMatrix:rightMatrix:result↩
Matrix:
MPSMatrixMultiplication, 109
encodeToCommandBuffer:primaryTexture:inPlace↩
SecondaryTexture:fallbackCopyAllocator:
MPSBinaryImageKernel, 18
encodeToCommandBuffer:primaryTexture:secondary↩
Texture:destinationTexture:
MPSBinaryImageKernel, 19
encodeToCommandBuffer:sourceImage:destination↩
Image:
MPSCNNKernel, 33
encodeToCommandBuffer:sourceTexture:destination↩
Texture:
MPSUnaryImageKernel, 118
encodeToCommandBuffer:sourceTexture:histogram↩
:histogramOffset:
MPSImageHistogram, 73
encodeTransformToCommandBuffer:sourceTexture↩
:histogram:histogramOffset:
MPSImageHistogramEqualization, 75
encodeTransformToCommandBuffer:sourceTexture↩
:sourceHistogram:sourceHistogramOffset↩
:desiredHistogram:desiredHistogramOffset:
MPSImageHistogramSpecification, 78

featureChannels
MPSImage, 56
MPSImageDescriptor, 67

groups
MPSCNNConvolution, 24
MPSCNNConvolutionDescriptor, 26

height
MPSImage, 56

MPSImageDescriptor, 67
MPSSize, 113
histogramForAlpha
MPSImageHistogramInfo, 77
histogramInfo
MPSImageHistogram, 74
MPSImageHistogramEqualization, 76
MPSImageHistogramSpecification, 79
histogramSizeForSourceFormat:
MPSImageHistogram, 73

imageDescriptorWithChannelFormat:width:height:featureChannels:
MPSImageDescriptor, 66
imageDescriptorWithChannelFormat:width:height:featureChannels:numberOfImages:usage:
MPSImageDescriptor, 66
initWithBuffer:descriptor:
MPSMatrix, 105
initWithDevice:
MPSCNNConvolution, 22
MPSCNNCrossChannelNormalization, 28
MPSCNNFullyConnected, 30
MPSCNNLocalContrastNormalization, 36
MPSCNNNeuronAbsolute, 39
MPSCNNNeuronLinear, 40
MPSCNNNeuronReLU, 42
MPSCNNNeuronSigmoid, 44
MPSCNNNeuronTanH, 45
MPSCNNPooling, 46
MPSCNNSpatialNormalization, 51
MPSImageAreaMax, 58
MPSImageBox, 61
MPSImageDilate, 68
MPSImageGaussianBlur, 71
MPSImageMedian, 84
MPSImagePyramid, 86
MPSImageSobel, 88
MPSImageThresholdBinary, 90
MPSImageThresholdBinaryInverse, 93
MPSImageThresholdToZero, 94
MPSImageThresholdToZeroInverse, 96
MPSImageThresholdTruncate, 97
MPSKernel, 101
MPSMatrixMultiplication, 109
initWithDevice:a:
MPSCNNNeuronReLU, 43
initWithDevice:a:b:
MPSCNNNeuronLinear, 40
MPSCNNNeuronTanH, 45
initWithDevice:centerWeight:
MPSImagePyramid, 86
initWithDevice:convolutionDescriptor:kernelWeights:biasTerms:flags:
MPSCNNConvolution, 22
MPSCNNFullyConnected, 30
initWithDevice:histogramInfo:
MPSImageHistogram, 74
MPSImageHistogramEqualization, 76
MPSImageHistogramSpecification, 79
initWithDevice:imageDescriptor:
MPSImage, 55
MPSTemporaryImage, 114
initWithDevice:kernelDiameter:
MPSImageMedian, 84
initWithDevice:kernelSize:
MPSCNNCrossChannelNormalization, 29
initWithDevice:kernelWidth:kernelHeight:
MPSCNNLocalContrastNormalization, 36
MPSCNNPooling, 46
MPSCNNSpatialNormalization, 51
MPSImageAreaMax, 58
MPSImageBox, 61
initWithDevice:kernelWidth:kernelHeight:strideInPixelsX:strideInPixelsY:
MPSCNNPooling, 48
initWithDevice:kernelWidth:kernelHeight:values:
MPSImageDilate, 69
initWithDevice:kernelWidth:kernelHeight:weights:
MPSImageConvolution, 65
MPSImagePyramid, 86
initWithDevice:linearGrayColorTransform:
MPSImageSobel, 88
initWithDevice:sigma:
MPSImageGaussianBlur, 71
initWithDevice:srcAlpha:destAlpha:backgroundColor:conversionInfo:
MPSImageConversion, 63
initWithDevice:thresholdValue:linearGrayColorTransform:
MPSImageThresholdToZero, 94
MPSImageThresholdToZeroInverse, 96
MPSImageThresholdTruncate, 97
initWithDevice:thresholdValue:maximumValue:linearGrayColorTransform:
MPSImageThresholdBinary, 90
MPSImageThresholdBinaryInverse, 93
initWithDevice:transposeLeft:transposeRight:resultRows:resultColumns:interiorColumns:alpha:beta:
MPSMatrixMultiplication, 109
initWithTexture:featureChannels:
MPSImage, 55
MPSTemporaryImage, 114
inputFeatureChannels
MPSCNNConvolution, 24
MPSCNNConvolutionDescriptor, 26

kernelDiameter
MPSImageMedian, 85
kernelHeight
MPSCNNConvolution, 24
MPSCNNConvolutionDescriptor, 26
MPSCNNLocalContrastNormalization, 37
MPSCNNPooling, 48
MPSCNNSpatialNormalization, 53
MPSImageAreaMax, 60
MPSImageBox, 62

MPSImageConvolution, 65
MPSImageDilate, 69
MPSImagePyramid, 87
kernelSize
MPSCNNCrossChannelNormalization, 29
kernelWidth
MPSCNNConvolution, 24
MPSCNNConvolutionDescriptor, 26
MPSCNNLocalContrastNormalization, 37
MPSCNNPooling, 48
MPSCNNSpatialNormalization, 53
MPSImageAreaMax, 60
MPSImageBox, 62
MPSImageConvolution, 65
MPSImageDilate, 69
MPSImagePyramid, 87

label
MPSImage, 56
MPSKernel, 104
leftMatrixOrigin
MPSMatrixMultiplication, 110

MPSBinaryImageKernel, 17
clipRect, 20
encodeToCommandBuffer:inPlacePrimary↩
Texture:secondaryTexture:fallbackCopy↩
Allocator:, 18
encodeToCommandBuffer:primaryTexture:in↩
PlaceSecondaryTexture:fallbackCopy↩
Allocator:, 18
encodeToCommandBuffer:primaryTexture↩
:secondaryTexture:destinationTexture:, 19
primaryEdgeMode, 20
primaryOffset, 21
primarySourceRegionForDestinationSize:, 19
secondaryEdgeMode, 21
secondaryOffset, 21
secondarySourceRegionForDestinationSize:, 20
MPSCNNConvolution, 21
groups, 24
initWithDevice:, 22
initWithDevice:convolutionDescriptor:kernel↩
Weights:biasTerms:flags:, 22
inputFeatureChannels, 24
kernelHeight, 24
kernelWidth, 24
neuron, 24
outputFeatureChannels, 24
strideInPixelsX, 24
strideInPixelsY, 24
MPSCNNConvolutionDescriptor, 25
cnnConvolutionDescriptorWithKernelWidth↩
:kernelHeight:inputFeatureChannels:output↩
FeatureChannels:neuronFilter:, 25
groups, 26
inputFeatureChannels, 26
kernelHeight, 26
kernelWidth, 26
neuron, 26
outputFeatureChannels, 26
strideInPixelsX, 26
strideInPixelsY, 26
MPSCNNCrossChannelNormalization, 27
alpha, 29
beta, 29
delta, 29
initWithDevice:, 28
initWithDevice:kernelSize:, 29
kernelSize, 29
MPSCNNFullyConnected, 29
initWithDevice:, 30
initWithDevice:convolutionDescriptor:kernel↩
Weights:biasTerms:flags:, 30
MPSCNNKernel, 32
clipRect, 34
destinationFeatureChannelOffset, 34
edgeMode, 35
encodeToCommandBuffer:sourceImage:destination↩
Image:, 33
offset, 35
sourceRegionForDestinationSize:, 34
MPSCNNLocalContrastNormalization, 35
alpha, 37
beta, 37
delta, 37
initWithDevice:, 36
initWithDevice:kernelWidth:kernelHeight:, 36
kernelHeight, 37
kernelWidth, 37
p0, 37
pm, 37
ps, 37
MPSCNNLogSoftMax, 37
MPSCNNNeuron, 38
MPSCNNNeuronAbsolute, 39
initWithDevice:, 39
MPSCNNNeuronLinear, 39
initWithDevice:, 40
initWithDevice:a:b:, 40
MPSCNNNeuronReLU, 41
initWithDevice:, 42
initWithDevice:a:, 43
MPSCNNNeuronSigmoid, 43
initWithDevice:, 44
MPSCNNNeuronTanH, 44
initWithDevice:, 45
initWithDevice:a:b:, 45
MPSCNNPooling, 45
initWithDevice:, 46
initWithDevice:kernelWidth:kernelHeight:, 46
initWithDevice:kernelWidth:kernelHeight:strideIn↩
PixelsX:strideInPixelsY:, 48
kernelHeight, 48
kernelWidth, 48
strideInPixelsX, 48
strideInPixelsY, 48

MPSCNNPoolingAverage, 48
MPSCNNPoolingMax, 49
MPSCNNSoftMax, 50
MPSCNNSpatialNormalization, 50
alpha, 53
beta, 53
delta, 53
initWithDevice:, 51
initWithDevice:kernelWidth:kernelHeight:, 51
kernelHeight, 53
kernelWidth, 53
MPSCopyAllocator
MPSKernel, 103
MPSImage, 53
device, 56
featureChannels, 56
height, 56
initWithDevice:imageDescriptor:, 55
initWithTexture:featureChannels:, 55
label, 56
numberOfImages, 56
pixelFormat, 56
pixelSize, 56
precision, 57
setPurgeableState:, 56
texture, 57
textureType, 57
usage, 57
width, 57
MPSImageAreaMax, 57
initWithDevice:, 58
initWithDevice:kernelWidth:kernelHeight:, 58
kernelHeight, 60
kernelWidth, 60
MPSImageAreaMin, 60
MPSImageBox, 61
initWithDevice:, 61
initWithDevice:kernelWidth:kernelHeight:, 61
kernelHeight, 62
kernelWidth, 62
MPSImageConversion, 62
destinationAlpha, 63
initWithDevice:srcAlpha:destAlpha:background↩
Color:conversionInfo:, 63
sourceAlpha, 63
MPSImageConvolution, 64
bias, 65
initWithDevice:kernelWidth:kernelHeight:weights:, 65
kernelHeight, 65
kernelWidth, 65
MPSImageDescriptor, 65
channelFormat, 67
cpuCacheMode, 67
featureChannels, 67
height, 67
imageDescriptorWithChannelFormat:width↩
:height:featureChannels:, 66
imageDescriptorWithChannelFormat:width↩
:height:featureChannels:numberOfImages↩
:usage:, 66
numberOfImages, 67
pixelFormat, 67
storageMode, 67
usage, 67
width, 67
MPSImageDilate, 67
initWithDevice:, 68
initWithDevice:kernelWidth:kernelHeight:values:, 69
kernelHeight, 69
kernelWidth, 69
MPSImageErode, 69
MPSImageGaussianBlur, 70
initWithDevice:, 71
initWithDevice:sigma:, 71
sigma, 72
MPSImageGaussianPyramid, 72
MPSImageHistogram, 72
clipRectSource, 74
encodeToCommandBuffer:sourceTexture:histogram↩
:histogramOffset:, 73
histogramInfo, 74
histogramSizeForSourceFormat:, 73
initWithDevice:histogramInfo:, 74
zeroHistogram, 74
MPSImageHistogramEqualization, 74
encodeTransformToCommandBuffer:source↩
Texture:histogram:histogramOffset:, 75
histogramInfo, 76
initWithDevice:histogramInfo:, 76
MPSImageHistogramInfo, 76
histogramForAlpha, 77
maxPixelValue, 77
minPixelValue, 77
numberOfHistogramEntries, 77
MPSImageHistogramSpecification, 77
encodeTransformToCommandBuffer:source↩
Texture:sourceHistogram:sourceHistogram↩
Offset:desiredHistogram:desiredHistogram↩
Offset:, 78
histogramInfo, 79
initWithDevice:histogramInfo:, 79
MPSImageIntegral, 80
MPSImageIntegralOfSquares, 80
MPSImageLanczosScale, 81
scaleTransform, 82
MPSImageLaplacian, 82
bias, 83
MPSImageMedian, 83
initWithDevice:, 84
initWithDevice:kernelDiameter:, 84
kernelDiameter, 85
maxKernelDiameter, 84
minKernelDiameter, 85
MPSImagePyramid, 85

initWithDevice:, 86
initWithDevice:centerWeight:, 86
initWithDevice:kernelWidth:kernelHeight:weights:, 86
kernelHeight, 87
kernelWidth, 87
MPSImageSobel, 87
colorTransform, 88
initWithDevice:, 88
initWithDevice:linearGrayColorTransform:, 88
MPSImageTent, 89
MPSImageThresholdBinary, 89
initWithDevice:, 90
initWithDevice:thresholdValue:maximumValue:linearGrayColorTransform:, 90
maximumValue, 92
thresholdValue, 92
transform, 92
MPSImageThresholdBinaryInverse, 92
initWithDevice:, 93
initWithDevice:thresholdValue:maximumValue:linearGrayColorTransform:, 93
maximumValue, 93
thresholdValue, 93
transform, 93
MPSImageThresholdToZero, 94
initWithDevice:, 94
initWithDevice:thresholdValue:linearGrayColorTransform:, 94
thresholdValue, 95
transform, 95
MPSImageThresholdToZeroInverse, 95
initWithDevice:, 96
initWithDevice:thresholdValue:linearGrayColorTransform:, 96
thresholdValue, 96
transform, 96
MPSImageThresholdTruncate, 97
initWithDevice:, 97
initWithDevice:thresholdValue:linearGrayColorTransform:, 97
thresholdValue, 99
transform, 99
MPSImageTranspose, 99
MPSKernel, 100
copyWithZone:device:, 101
device, 104
initWithDevice:, 101
label, 104
MPSCopyAllocator, 103
MPSRectNoClip, 104
options, 104
MPSMatrix, 104
columns, 106
data, 106
dataType, 106
device, 106
initWithBuffer:descriptor:, 105
rowBytes, 106
rows, 106
MPSMatrixDescriptor, 106
columns, 107
dataType, 107
matrixDescriptorWithDimensions:columns:rowBytes:dataType:, 107
rowBytes, 107
rowBytesFromColumns:dataType:, 107
rows, 107
MPSMatrixMultiplication, 108
encodeToCommandBuffer:leftMatrix:rightMatrix:resultMatrix:, 109
initWithDevice:, 109
initWithDevice:transposeLeft:transposeRight:resultRows:resultColumns:interiorColumns:alpha:beta:, 109
leftMatrixOrigin, 110
resultMatrixOrigin, 110
rightMatrixOrigin, 110
MPSOffset, 110
x, 111
y, 111
z, 111
MPSOrigin, 111
x, 111
y, 111
z, 111
MPSRectNoClip
MPSKernel, 104
MPSRegion, 112
origin, 112
size, 112
MPSScaleTransform, 112
MPSSize, 113
depth, 113
height, 113
width, 113
MPSTemporaryImage, 113
initWithDevice:imageDescriptor:, 114
initWithTexture:featureChannels:, 114
prefetchStorageWithCommandBuffer:imageDescriptorList:, 115
temporaryImageWithCommandBuffer:imageDescriptor:, 115
temporaryImageWithCommandBuffer:textureDescriptor:, 115
MPSUnaryImageKernel, 116
clipRect, 119
edgeMode, 119
encodeToCommandBuffer:inPlaceTexture:fallbackCopyAllocator:, 117
encodeToCommandBuffer:sourceTexture:destinationTexture:, 118
offset, 119
sourceRegionForDestinationSize:, 118
matrixDescriptorWithDimensions:columns:rowBytes:dataType:

MPSMatrixDescriptor, 107
maxKernelDiameter
MPSImageMedian, 84
maxPixelValue
MPSImageHistogramInfo, 77
maximumValue
MPSImageThresholdBinary, 92
MPSImageThresholdBinaryInverse, 93
minKernelDiameter
MPSImageMedian, 85
minPixelValue
MPSImageHistogramInfo, 77

neuron
MPSCNNConvolution, 24
MPSCNNConvolutionDescriptor, 26
numberOfHistogramEntries
MPSImageHistogramInfo, 77
numberOfImages
MPSImage, 56
MPSImageDescriptor, 67

offset
MPSCNNKernel, 35
MPSUnaryImageKernel, 119
options
MPSKernel, 104
origin
MPSRegion, 112
outputFeatureChannels
MPSCNNConvolution, 24
MPSCNNConvolutionDescriptor, 26

p0
MPSCNNLocalContrastNormalization, 37
pixelFormat
MPSImage, 56
MPSImageDescriptor, 67
pixelSize
MPSImage, 56
pm
MPSCNNLocalContrastNormalization, 37
precision
MPSImage, 57
prefetchStorageWithCommandBuffer:imageDescriptor↩
List:
MPSTemporaryImage, 115
primaryEdgeMode
MPSBinaryImageKernel, 20
primaryOffset
MPSBinaryImageKernel, 21
primarySourceRegionForDestinationSize:
MPSBinaryImageKernel, 19
ps
MPSCNNLocalContrastNormalization, 37

resultMatrixOrigin
MPSMatrixMultiplication, 110
rightMatrixOrigin
MPSMatrixMultiplication, 110
rowBytes
MPSMatrix, 106
MPSMatrixDescriptor, 107
rowBytesFromColumns:dataType:
MPSMatrixDescriptor, 107
rows
MPSMatrix, 106
MPSMatrixDescriptor, 107

scaleTransform
MPSImageLanczosScale, 82
secondaryEdgeMode
MPSBinaryImageKernel, 21
secondaryOffset
MPSBinaryImageKernel, 21
secondarySourceRegionForDestinationSize:
MPSBinaryImageKernel, 20
setPurgeableState:
MPSImage, 56
sigma
MPSImageGaussianBlur, 72
size
MPSRegion, 112
sourceAlpha
MPSImageConversion, 63
sourceRegionForDestinationSize:
MPSCNNKernel, 34
MPSUnaryImageKernel, 118
storageMode
MPSImageDescriptor, 67
strideInPixelsX
MPSCNNConvolution, 24
MPSCNNConvolutionDescriptor, 26
MPSCNNPooling, 48
strideInPixelsY
MPSCNNConvolution, 24
MPSCNNConvolutionDescriptor, 26
MPSCNNPooling, 48

temporaryImageWithCommandBuffer:imageDescriptor↩
:
MPSTemporaryImage, 115
temporaryImageWithCommandBuffer:textureDescriptor↩
:
MPSTemporaryImage, 115
texture
MPSImage, 57
textureType
MPSImage, 57
thresholdValue
MPSImageThresholdBinary, 92
MPSImageThresholdBinaryInverse, 93
MPSImageThresholdToZero, 95
MPSImageThresholdToZeroInverse, 96
MPSImageThresholdTruncate, 99
transform
MPSImageThresholdBinary, 92
MPSImageThresholdBinaryInverse, 93

MPSImageThresholdToZero, 95
MPSImageThresholdToZeroInverse, 96
MPSImageThresholdTruncate, 99

usage
MPSImage, 57
MPSImageDescriptor, 67

width
MPSImage, 57
MPSImageDescriptor, 67
MPSSize, 113

x
MPSOffset, 111
MPSOrigin, 111

y
MPSOffset, 111
MPSOrigin, 111

z
MPSOffset, 111
MPSOrigin, 111
zeroHistogram
MPSImageHistogram, 74